

Runtime Verification

Grigore Roşu

University of Illinois at Urbana-Champaign

Contents

1	Introduction	11
1.1	A Taxonomy of Runtime Analysis Techniques	21
1.1.1	Trace Storing versus Non-Storing Algorithms	21
1.1.2	Synchronous versus Asynchronous Monitoring	23
1.1.3	Predictive versus Exact Analysis	24
2	Background, Preliminaries, Notations	27
2.1	Preliminaries	30
2.1.1	Membership Equational Logic	30
2.1.2	Maude	34
3	Safety Properties	41
3.1	Finite Traces	44
3.2	Infinite Traces	51
3.3	Finite and Infinite Traces	54
3.4	“Always Past” Characterization	57
4	Monitoring	61
4.1	Specifying Safety Properties as Monitors	61
4.2	Complexity of Monitoring a Safety Property	66
4.3	Monitoring Safety Properties is Arbitrarily Hard	72
4.4	Canonical Monitors	74
5	Event/Trace Observation	77
6	Monitor Synthesis: Finite State Machines (FSM)	79
6.1	Binary Transition Trees (BTT)	79
6.1.1	Multi-Transition and Binary Transition Tree Finite State Machines	79

6.1.2	From MT-FSMs to BTT-FSMs	84
6.2	Multi-Transitions and Binary Transition Trees	87
6.3	Binary Transition Tree Finite State Machines	90
7	Monitor Synthesis: Extended Regular Expressions (ERE)	93
7.1	Monitoring ERE Safety Needs Non-Elementary Space	93
7.1.1	Discussion and Relevance of the Lower-Bound Result	94
7.1.2	The Lower-Bound Result	100
7.2	Generating Optimal Monitors for ERE	106
7.2.1	Introduction	107
7.2.2	Extended Regular Expressions and Derivatives	110
7.2.3	Hidden Logic and Coinduction	112
7.2.4	Behavioral Equivalence, Satisfaction and Specification	113
7.2.5	Generating Minimal DFA Monitors by Coinduction	119
7.2.6	Implementation and Evaluation	121
8	Monitor Synthesis: Linear Temporal Logic (LTL)	127
8.1	Finite Trace Future Time Linear Temporal Logic	127
8.1.1	Finite Trace Semantics	128
8.1.2	Finite Trace Semantics in Maude	129
8.2	A Backwards, Asynchronous, but Efficient Algorithm	131
8.2.1	An Example	131
8.2.2	Generating Dynamic Programming Algorithms	134
8.3	A Forwards and Often Synchronous Algorithm	137
8.3.1	An Event Consuming Algorithm	137
8.3.2	Correctness and Completeness	142
8.3.3	Further Optimization by Memoization	144
8.4	Generating Forwards, Synchronous and Efficient Monitors	146
8.4.1	From LTL Formulae to BTT-FSMs	147
8.4.2	Examples	152
8.5	Conclusions	154
9	Efficient Monitoring of ω-Languages	157
9.1	Introduction	157
9.2	Preliminaries: Büchi Automata	161
9.3	Generating a <i>BTT-FSM</i> from a Büchi automaton	162
9.4	Monitor Generation and MOP	166
9.4.1	Evaluation	167
9.5	Conclusions	168

10 Efficient Monitoring of “Always Past” Temporal Safety	175
10.1 Introduction	176
10.2 The PathExplorer Architecture	179
10.2.1 The Observer	179
10.2.2 Code Instrumentation	180
10.3 Finite Trace Past Time LTL	181
10.3.1 Syntax	182
10.3.2 Formal Semantics	183
10.3.3 Recursive Semantics	184
10.3.4 Equivalent Logics	185
10.4 Monitoring Safety by Rewriting	187
10.4.1 Maude	188
10.4.2 Formulae and Data Structures	189
10.4.3 Propositional Calculus	190
10.4.4 Past Time Linear Temporal Logic	191
10.4.5 Monitoring with Maude	195
10.5 Synthesizing Monitors for Safety Properties	196
10.5.1 The Algorithm Illustrated by an Example	197
10.5.2 The Algorithm Formalized	200
10.5.3 Optimizing the Generated Code	204
10.5.4 Implementation of Offline and Inline Monitoring	206
10.6 Conclusion	211
10.7 Optimal Monitoring of “Always Past” Temporal Safety	212
10.7.1 The Monitor Synthesis Algorithm	212
10.7.2 A Maude Implementation of the Monitor Synthesizer	215
11 Monitoring “Always-Past” Temporal Safety with Call-Return	223
11.1 Introduction	224
11.2 PTLTL and PTCARET	226
11.3 PTCARET Derived Operators and Examples	232
11.4 A Monitor Synthesis Algorithm for PTCARET	234
11.4.1 The Target Language	235
11.4.2 The Monitor Synthesis Algorithm	237
11.4.3 Implementation as Logic Plugin, Optimizations, Example	241
11.5 Conclusion and Future Work	245
11.6 Auxiliary Material - not included in original paper	246
11.6.1 The Algorithm Formalized	249

12 Efficient Monitoring of Parametric Context-Free Patterns	255
12.1 Introduction	255
12.1.1 Example	256
12.1.2 Contributions	260
12.1.3 Paper Outline	262
12.2 Related Work	262
12.2.1 Runtime Monitoring	262
12.2.2 Context-free Grammars in Testing and Verification . .	264
12.3 MOP Revisited	265
12.3.1 MOP in a Nutshell	266
12.4 Suffix Matching in JavaMOP	267
12.5 Context-Free Patterns in JavaMOP	270
12.5.1 Preliminaries	271
12.5.2 CFG Monitoring Algorithms	272
12.5.3 CFG-plug-in Implementation	278
12.5.4 Proofs of Correctness	279
12.6 Evaluation	284
12.6.1 Experimental Settings	285
12.6.2 Properties	285
12.6.3 Results	286
12.7 Conclusions and Future Work	292
13 Efficient Monitoring with Deterministic String Rewrite Sys-	
tems	295
13.1 Introduction	296
13.1.1 Examples	297
13.1.2 Contributions	300
13.1.3 Paper Outline	301
13.2 Related Work and MOP	301
13.3 Monitoring SRS Specifications	303
13.3.1 Preliminaries	304
13.3.2 String Rewriting Algorithm Overview	304
13.3.3 Pattern Match Automata	305
13.3.4 Rewriting using Pattern Match Automata	311
13.4 Evaluation	317
13.5 Conclusion	320

14 Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis	323
14.1 Introduction	324
14.2 Allen (Linear) Temporal Logic - ATL (ALTL)	325
14.3 Linear Translation of ALTL into <i>LTL</i>	330
14.4 Monitoring ALTL	333
14.5 Experiment	339
14.6 Conclusion	341
15 Parametric Property Monitoring	343
15.1 Introduction and Motivation	344
15.1.1 Motivating examples and highlights	344
15.1.2 Related Work	349
15.1.3 Contributions	351
15.1.4 Paper Structure	352
15.2 Examples of Parametric Properties	352
15.2.1 Releasing acquired resources	353
15.2.2 Authenticate before use	355
15.2.3 Safe iterators	356
15.2.4 Correct locking	357
15.2.5 Safe resource use by safe client	358
15.2.6 Success ratio	359
15.3 Mathematical Background: Partial Functions, Least Upper Bounds (lubs) and lub Closures	359
15.3.1 Partial Functions	360
15.3.2 Least Upper Bounds of Families of Sets of Partial Maps	363
15.3.3 Least Upper Bound Closures	366
15.4 Parametric Traces and Properties	370
15.4.1 The Non-Parametric Case	371
15.4.2 The Parametric Case	374
15.5 Slicing With Less	377
15.6 Algorithm for Online Parametric Trace Slicing	380
15.7 Monitors and Parametric Monitors	387
15.7.1 The Non-Parametric Case	387
15.7.2 The Parametric Case	390
15.8 Algorithms for Parametric Trace Monitoring	391
15.8.1 Unoptimized but Simpler Algorithm	391
15.8.2 Optimized Algorithm	394
15.9 Implementation in JavaMOP and RV	400

15.9.1	Implementation Optimizations	401
15.9.2	Experiments and Evaluation	403
15.10	Concluding Remarks, Future Work and Acknowledgments . .	408
16	Efficient Formalism-Independent Monitoring of Parametric Properties	411
16.1	Introduction	412
16.2	Approach Overview	416
16.3	Background: Parametric Monitoring	420
16.3.1	Events, Traces and Properties	421
16.3.2	Parametric Monitors	424
16.4	Monitoring with Creation Events : $\mathbb{C}^+\langle X \rangle$	426
16.5	Limitations of $\mathbb{C}^+\langle X \rangle$ and Enable Sets	430
16.5.1	Enable Sets	433
16.5.2	Computing Enable Sets	435
16.6	Monitoring with Enable Sets: $\mathbb{D}\langle X \rangle$	439
16.6.1	Timestamping Monitors: Algorithm $\mathbb{D}\langle X \rangle$	440
16.6.2	Proofs of Correctness	444
16.7	Implementation and Evaluation	449
16.8	Conclusion	456
17	Garbage Collection for Monitoring Parametric Properties	457
17.1	Introduction	458
17.2	Parametric Properties and Monitoring	464
17.3	Coenable Sets	469
17.4	Implementation	473
17.4.1	Indexing Trees	473
17.4.2	Collecting Unnecessary Monitors	474
17.5	Evaluation of the RV System	479
17.5.1	Experimental Settings	479
17.5.2	Results and Discussions	481
17.6	Conclusion	483
18	Predictive Runtime Analysis	485
19	Maximal Causal Models for Sequentially Consistent Systems	487
19.1	Introduction	488
19.1.1	Motivating Examples	489

19.2	Execution Model	491
19.2.1	Concurrent Objects, Serial Specification	491
19.2.2	Events and Traces	492
19.3	Feasibility Model	493
19.4	Maximality	496
19.5	Proving Soundness of Existing Causal Models	504
19.5.1	Happens Before Relation on Mazurkiewicz Traces . . .	504
19.5.2	Weak Happens Before	505
19.5.3	Happens-Before with synchronization	506
19.5.4	Weak-Happens-Before with synchronization	507
19.6	Characterizing the Feasibility Closure	508
19.7	Model Checking the Feasibility Closure	511
19.8	Related Work and Discussion	513
19.9	Conclusion	515
20	Static Analysis to Improve Runtime Verification	517
21	Semantics-Based Runtime Verification	519
21.1	Defining a Formal Semantics	519
21.2	Semantics-Based Symbolic Execution	519
21.3	Program Verification as Exhaustive Runtime Verification . . .	519
22	Conclusion and Future Work	521
22.1	Safety Properties and Monitoring	521

Topics to cover:

- Safety properties and their monitoring. How many safety properties are there? Can they all be monitored? Complexity of monitoring in different formalism: LTL, RE, ERE, CFG, SRS. Both dynamic properties, like above, and static properties (e.g., complex heap patterns).
- Event/Trace observation. How to observe the execution of a program? Instrumentation vs. runtime environment.
- Monitor synthesis. Generating optimal monitors for several formalisms: LTL, FT/PT-LTL, RE, ERE, CFG, SRS, PT-CaRet, Allen TL.
- Parametric property monitoring. How to deal with multiple instances of monitors?
- Predictive runtime analysis. Vector clock vs SMT-based techniques.
- Static analysis to improve runtime verification. Improve runtime overhead by not instrumenting what is unnecessary. Improve prediction capability by looking beyond the trace.
- Semantics-based Runtime Verification. Defining a formal language semantics. Using a semantics to do symbolic execution and runtime verify properties. Ultimate goal: verify programs by exhaustive runtime verification.

Papers which have been completely absorbed:

- * [241]: Section 7.2; Chapter 2
- * [227]: Chapter 2; Chapter 3; Chapter 4; Section 7.1;
- * [223]: Section 6.1; Chapter 8;
- * [84]: Chapter 6; Chapter 9;
- * [139, 138]: Chapter 10;
- * [229]: Chapter 11

Chapter 1

Introduction

get citations from papers sent by Klaus on March 26, 2017. Also ask the RV list.

Begin of intro stuff for chapter on safety

From SACS: Abstract sacs: *This paper addresses the problem of runtime verification from a foundational perspective, answering questions like “Is there a consensus among the various definitions of a safety property?” (Answer: Yes), “How many safety properties exist?” (Answer: As many as real numbers), “How difficult is the problem of monitoring a safety property?” (Answer: Arbitrarily complex), “Is there any formalism that can express all safety properties?” (Answer: No), etc. Various definitions of safety properties as sets of execution traces have been proposed in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments and a novel notion of persistence, this paper first establishes the existence of bijective correspondences between the various notions of safety property. It then shows that safety properties can be characterized as “always past” properties. Finally, it proposes a general notion of monitor, which allows to show that safety properties correspond precisely to the monitorable properties, and then to establish that monitoring a safety property is arbitrarily hard.*

From safety: *Abstract safety:* Various definitions of safety properties as sets of execution traces have been introduced in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments, this paper first shows that these notions of safety are ultimately equivalent, by showing each of them to have the cardinal of the continuum. It is then shown that all safety properties can be characterized as “always past” properties, and then that the problem of monitoring a safety property can be arbitrarily hard. Finally, two decidable specification formalisms for safety properties are discussed, namely extended regular expressions and past time LTL. It is shown that monitoring the former requires non-elementary space. An optimal monitor synthesis algorithm is given for the latter; the generated monitors run in space linear with the number of temporal operators and in time linear with the size of the formula.

A *safety property* is a behavioral property which, once violated, cannot be satisfied anymore. For example, a property “always $x > 0$ ” is violated when $x \leq 0$ is observed for the first time; this safety property remains violated even though eventually $x > 0$ might hold. That means that one can identify each safety property with a set of “bad” finite execution traces, with the intuition that once one of those is reached the safety property is violated.

There are several apparently different ways to formalize safety. Perhaps the most immediate one is to complement the “bad traces” above and thus to define a safety property as a prefix-closed property over finite traces (containing the “good traces”) – by “property” in this paper we mean a set of finite or infinite traces. Inspired by Lamport [180], Alpern and Schneider [11] define safety properties over infinite traces as ones with the property that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable, in the sense that there is no acceptable infinite completion of it. Is there any relationship between these two definitions of safety? We show rather indirectly that there is, by showing that their corresponding sets of safety properties have the cardinal c of the continuum (i.e., the cardinal of \mathbb{R} , the set of real numbers), so there exists some bijective mapping between the two. Unfortunately, the existence of such a bijection is as little informative as the existence of a bijection between the real numbers and the irrational numbers. To capture the relationship

between finite- and infinite-trace safety properties in a meaningful way, we introduce a subset of finite-trace safety properties, called *persistent*, and then construct an explicit bijection between that subset and the infinite-trace safety properties. Interestingly, over finite traces there are as many safety properties as unrestricted properties (finite-traces are enumerable and $\mathcal{P}(\mathbb{N})$ is in bijection with \mathbb{R}), while over infinite traces there are c safety properties versus 2^c unrestricted properties (infinite traces are in bijection with \mathbb{R}).

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [181] extends the notion of infinite-trace safety properties to properties over both finite and infinite traces, while Schneider et al. [235, 123] give an alternative definition of safety over finite and infinite traces, called “execution monitoring”. One immediate technical advantage of allowing both finite and infinite traces is that one can define prefix-closed properties. We indirectly show that prefix-closeness is not a sufficient condition to define safety properties when infinite traces are also allowed, by showing that there are 2^c prefix-closed properties versus, as expected, “only” c safety properties.

Another common way to specify safety properties is as “always past” properties, that is, as properties containing only words whose finite prefixes satisfy a given property. If P is a property on finite prefixes, then we write $\Box P$ for the “always P ” safety property containing the words with prefixes in P . We show that specifying safety properties as “always past” properties is fully justified by showing that, for each of the three types of traces (finite, infinite, and both), the “always past” properties are precisely the safety properties as defined above. It is common to specify P using some logical formalism, for example past time linear temporal logic (past LTL) [187]; for example, one can specify “ a before b ” in past LTL as the formula $b \rightarrow \Diamond a$.

The problem of monitoring safety properties is also investigated in this paper. Since there are as many safety properties as real numbers, it is not unexpected that some of them can be very hard to monitor. We show that the problem of monitoring a safety property is arbitrarily hard, by showing that it reduces to deciding membership of natural numbers to a set of natural numbers. In particular, we can associate a safety property to any degree in the arithmetic hierarchy as well as to any complexity class in the decidable universe, whose monitoring is as hard as that degree or complexity class.

From SACS: *This paper makes three novel contributions, two technical and another pedagogical. On the technical side, it first introduces the notion of a persistent safety property, which appears to be the right finite-trace correspondent of an infinite-trace safety property, and uses it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Also on the technical side, it rigorously defines the problem of monitoring a safety property, and it shows that it can be arbitrarily hard. On the pedagogical side, this paper offers the first comprehensive study and uniform presentation of safety properties and of their monitoring.*

From safety: *In practice not all ($c = |\mathbb{R}|$) safety properties are meaningful, but only those ($\aleph_0 = |\mathbb{N}|$) which are specifiable using formal specification languages or logics of interest. We also investigate the problem of monitoring safety properties expressed using two common formalisms, namely regular expressions extended with complement, also called extended regular expressions (ERE), and LTL. It is known that both formalisms allow polynomial finite-trace membership checking algorithms [148, 223] if one has random access to the trace, but that both require exponential space if the trace can only be analyzed online [219, 172]. It is also known that LTL can indeed be monitored in exponential space [85] and so is claimed¹ for EREs in [219]. We show that the claim in [219] is, unfortunately, wrong, by showing that ERE monitoring requires non-elementary space. To do so, we propose for any $n \in \mathbb{N}$ a safety property P_n whose monitoring requires space non-elementary in n , as well as an ERE of size $O(n^3)$. Since the known monitoring algorithms for LTL in its full generality are asymptotically optimal, what is left to do is to consider important fragments of LTL. We focus on the “always past” fragment and give a monitor synthesis algorithm that takes formulae φ and generate monitors for them that need $O(k)$ total space and $O(|\varphi|)$ time to process each event, where k is the number of past operators in φ . This improves over the best known algorithm that needs space $O(|\varphi|)$ (and same time).*

from J.ASE 2005: Techniques for efficiently evaluating future time Linear Temporal Logic (abbreviated LTL) formulae on finite execution traces are presented. While the standard models of LTL are infinite traces, finite traces appear naturally when testing and/or monitoring real applications that only run for limited time periods. A finite trace variant of LTL is formally defined, together with an immediate executable semantics which turns out to be quite inefficient if used directly, via rewriting, as a monitoring procedure. Then three algorithms are investigated. First, a simple synthesis algorithm for monitors based on dynamic programming is presented; despite the efficiency of the generated monitors, they unfortunately need to analyze the trace backwards, thus making them unusable in most practical situations. To circumvent this problem, two rewriting-based practical algorithms are further investigated, one using rewriting directly as a means for online monitoring, and the other using rewriting to generate automata-like monitors, called binary transition tree finite state machines (and abbreviated BTT-FSMs). Both rewriting algorithms are implemented in Maude, an executable specification language based on a very efficient implementation of term rewriting. The first rewriting algorithm essentially consists of a set of equations establishing an executable semantics of LTL, using a simple formula transforming approach. This algorithm is further improved to build automata on-the-fly via caching and reuse of rewrites (called memoization), resulting in a very efficient and small Maude program that can be used to monitor program executions. The second rewriting algorithm builds on the first one and synthesizes provably minimal BTT-FSMs from LTL formulae, which can then be used to analyze execution traces online without the need for a rewriting system. The presented work is part of an ambitious runtime verification and monitoring project at NASA Ames, called PATHEXPLORER, and demonstrates that rewriting can be a tractable and attractive means for experimenting and implementing logics for program monitoring.

Future time Linear Temporal Logic, abbreviated LTL, was introduced by Pnueli in 1977 [212] (see also [186, 188]) for stating properties about reactive and concurrent systems. LTL provides temporal operators that refer to the future/remaining part of an execution trace relative to a current point of reference. The standard models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests. Methods, such as model checking, have been developed for proving programs correct with respect to requirements specified as LTL formulae. Several systems are currently being developed that apply model checking to software systems written in Java, C and C++ [26, 86, 146, 78, 211, 112, 256, 134, 265]. However, for very large systems, there is little hope that one can actually prove correctness, and one must in those cases rely on debugging and testing. In the context of highly reliable and/or safety critical systems, one would actually want to *monitor* a program execution during operation and to determine whether it conforms to its specification. Any violation of the specification can then be used to guide the execution of the program into a safe state, either manually or automatically. In this paper we describe a collection of algorithms for monitoring program executions against LTL formulae. It is demonstrated how term rewriting, and in particular the Maude rewriting system [72], can be used to implement some of these algorithms very efficiently and conveniently.

The work presented in this paper has been started as part of, and stimulated by, the PATHEXPLORER project at NASA Ames, and in particular the Java PATHEXPLORER (JPAX) tool [135, 136] for monitoring Java programs. JPAX facilitates automated instrumentation of Java byte-code, currently using Compaq's JTREK which is not public anymore, but soon using BCEL [80]. The instrumented code emits relevant events to an observer during execution (see Figure 1.1). The observer can be running a Maude [72] process as a special case, so Maude's rewriting engine can be used to drive a temporal logic operational semantics with program execution events. The observer may run on a different computer, in which case the events are transmitted over a socket. The system is driven by a specification, stating what properties to be proved and what parts of the code to be instrumented. When the observer receives the events it dispatches these to a set of observer modules, each module performing a particular analysis that has been requested. In addition to checking temporal logic requirements, modules have also been

programmed to perform error pattern analysis of multi-threaded programs, predicting deadlock and data race potentials.

Figure 1.1: Overview of JPAX .

Using temporal logic in testing is an idea of broad practical and theoretical interest. One example is the commercial Temporal Rover and DBRover tools [87, 89], in which LTL properties are translated into code, which is then inserted at chosen positions in the program and executed whenever reached during program execution. The MaC tool [184, 165] is another example of a runtime monitoring tool. Here, Java byte-code is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. All the systems above try to discharge the program execution events as soon as possible, in order to minimize the space requirements. In contrast, a technique is proposed in [169] where the execution events are stored in an SQL database at runtime and then analyzed by means of queries after the program terminates. The PET tool, described in [121, 120, 119], uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Java MultiPathExplorer [240] is a tool which checks a past time LTL safety formula against a partial order extracted online from an execution trace. POTA [237] is another partial order trace analyzer system. Java-MoP [53] is a generic logic monitoring tool encouraging “monitoring-oriented programming” as a paradigm merging specification and implementation. Complexity results for testing a finite trace against temporal formulae expressed in different temporal logics are investigated in [191]. Algorithms using alternating automata to monitor LTL properties are proposed in [100], and a specialized LTL collecting statistics along the execution trace is described in [101]. Various algorithms to generate testing automata from temporal logic formulae are discussed in [214, 209], and [111] presents a Büchi automata inspired algorithm adapted to finite trace LTL.

The major goal of this paper is to present rewriting-based algorithms for effectively and efficiently evaluating LTL formulae on finite execution traces *online*, that is, by processing each event as it arrives. An important contribution of this paper is to show how a rewriting system, such as Maude, makes it possible to experiment with monitoring logics very efficiently and

elegantly, and furthermore can be used as a practical program monitoring engine. This approach allows one to formalize ideas in a framework close to standard mathematics. The presented algorithms are considered in the context of JPAX, but they can be easily adapted and used within other monitoring frameworks. We claim that the techniques presented in this paper, even though applied to LTL, are in fact generic and can be easily applied to other logics for monitoring. For example, in [219, 238] we applied the same generic, “formula transforming”, techniques to obtain rewriting based algorithms for situations in which the logic for monitoring was replaced by extended regular expressions (regular expressions with complement).

A non-trivial application of the rewriting based techniques presented in this paper is X9, a test-case generation and monitoring environment for a software system that controls the planetary NASA rover K9. This collaborative effort is described in more detail in [18] and it will be presented in full detail elsewhere soon. The rover controller, programmed in 35,000 lines of C++, essentially executes plans, where a plan is a tree-like structure consisting of actions and sub-actions. The leaf actions control various hardware on the rover, such as for example the camera and the wheels. The execution of a plan must cause the actions to be executed in the right order and must satisfy various time constraints, also part of the plan. Actions can start and eventually either terminate successfully or fail. Plans can specify how failed sub-actions can propagate upwards.

Testing the rover controller consists of generating plans and then monitoring that the plan actions are executed in the right order and that failures are propagated correctly. X9 automatically generates plans from a “grammar” of the structure of plans, using the Java PathFinder model checker [265]. For each plan, a set of temporal formulae that an execution of the plan must satisfy is also generated. For example, a plan may state that an action a should be executed by first executing a sub-action a_1 and then a sub-action a_2 , and that the failure of any of the sub-actions should not propagate: action a should eventually succeed, regardless of whether a_1 or a_2 fails. The generated temporal formulae will state these requirements, such as for example $\Box(\text{start}(a) \rightarrow \langle \rangle \text{succeed}(a))$ saying that “it is always the case (\Box) that when action a starts, then eventually ($\langle \rangle$) it terminates successfully”, and execution traces are monitored against them.

X9 is currently being turned into a mature system to be used by the developer. It is completely automated, generating a web-page containing all the warnings found. The top-level web-page identifies all the test-cases

that have failed (by violating some of the temporal properties), each linked to a web-page containing specifics such as the plan, the execution trace, and the properties that are violated. X9 has itself been tested by seeding errors into the rover controller code. The automated monitoring relieves the programmer from manually analyzing printed execution traces. Extending the logic with real-time, as is planned in future work, is crucial for this application since plans are heavily annotated with time constraints.

In Section 1.1, based on our experience, we give a rough classification of monitoring and runtime analysis algorithms by considering three important criteria. A first criterion is whether the execution trace of the monitored or analyzed program needs to be stored or not. Storing a trace might be very useful for specific types of analysis because one could have random access to events, but storing an execution trace is an expensive operation in practice, so sometimes trace-storing algorithms may not be desirable. A second criterion regards the synchronicity of the monitor, more precisely whether the monitor is able to react as soon as the specification or the requirement has been violated. Synchronicity may often trigger running a validity checker for the logic under consideration, which is typically a very expensive task. Finally, monitoring and analysis algorithms can also be classified as “predictive” versus “exact”, where the “exact” ones monitor the observed execution trace as a flat list of events, while the predictive algorithms try to guess potential erroneous behaviors of programs that can occur under different executions. All the algorithms in this paper are exact.

This paper requires a certain amount of mathematical notions and notations, which we introduce in Section 2.1 together with Maude [72], a high-performance system supporting both membership equational logic [201] and rewriting logic [200]. The current version of Maude can do more than 3 million rewritings per second on standard PCs, and its compiled version is intended to support more than 15 million rewritings per second², so it can quite well be used as an implementation language.

Section 8.1 defines the finite trace variant of linear temporal logic that we use in the rest of the paper. We found, by carefully analyzing several practical examples, that the most appropriate assumption to make at the end of the trace is that it is stationary in the last state. Then we define the semantics of the temporal operators using their usual meaning in infinite trace LTL, where the finite trace is infinitely extended by repeating the last state. Another option would be to consider that all atomic predicates are false or

²Personal communication by José Meseguer.

true in the state following the last one, but this would be problematic when inter-dependent predicates are involved, such as “gate-up” and “gate-down”.

In previous work we described a technique which synthesizes efficient dynamic programming algorithms for checking LTL formulae on finite execution traces [222]. Even though this algorithm is not dependent on rewriting (but it could be easily implemented in Maude by rewriting as we did with its dual variant for past time LTL [129, 53]), for the sake of completeness we present it in some detail in Section 8.2. This algorithm evaluates a formula bottom-up for each point in the trace, going backwards from the final state towards the initial state. Unfortunately, despite its linear complexity, this algorithm cannot be used online because it is both asynchronous and trace-storing. In [130, 131, 129] we dualize this technique and apply it to past time LTL, in which case the trace more naturally can be examined in a forwards direction synchronously.

Section 8.3 presents our first practical rewriting-based algorithm, which can directly monitor an LTL formula. This algorithm originates in [125, 222] and it was partially presented at the Automated Software Engineering conference [127]. The algorithm is expressed as a set of equations establishing an executable semantics of LTL using a simple formula transforming approach. The idea is to rewrite or transform an LTL monitoring requirement formula φ when an event e is received, to a formula $\varphi\{e\}$, which represents the new requirement that the monitored system should fulfill for the remaining part of the trace. This way, the LTL formula to monitor “evolves” into other LTL formulae by subsequent transformations. We show, however, that the size of the evolving formula is in the worst-case exponentially bounded by the size of the original LTL formula, and also that an exponential space explosion cannot be avoided in certain unfortunate cases. The efficiency of this rewriting algorithm can be improved by almost an order of magnitude by caching and reusing rewrites (a.k.a. “memoization”), which is supported by Maude. This algorithm is often synchronous, though there are situations in which it misses reporting a violation at the exact event when it occurs. The violation is, however, detected at a subsequent event. This algorithm can be relatively easily transformed into a synchronous one if one is willing to pay the price of running a validity checker, like the one presented in Subsection 8.4.1, after processing each event. The practical result of Section 8.3 is a very efficient and small Maude program that can be used to monitor program executions. The decision to use Maude has made it very easy to experiment with logics and algorithms in monitoring.

We finally present an alternative solution to monitoring LTL in Section 8.4, where a rewriting-based algorithm is used to *generate* an optimal special observer from an LTL formula. By optimality is meant everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) [46], whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM). BTT-FSMs can be used to analyze execution traces without the need for a rewriting system, and can hence be used by observers written in traditional programming languages. The BTT-FSM generator, which includes a validity checker, is also implemented in Maude and has about 200 lines of code in total.

1.1 A Taxonomy of Runtime Analysis Techniques

A *runtime analysis technique* is regarded in a broad sense in this section; it can be a method or a concrete algorithm that analyzes the execution trace of a running program and concludes a certain property about that program. Runtime analysis algorithms can be arbitrarily complex, depending upon the kind of properties to be monitored or analyzed. Based on our experience with current procedures implemented in JPaX, in this section we make an attempt to classify runtime analysis techniques. The three criteria below are intended to be neither exhaustive nor always applicable, but we found them quite useful in practice. They are not specific to any particular logic or approach, so we present them before we introduce our logic and algorithms. In fact, this taxonomy will allow us to appropriately discuss the benefits and drawbacks of our algorithms presented in the rest of the paper.

1.1.1 Trace Storing versus Non-Storing Algorithms

As events are received from the monitored system, a runtime analysis algorithm typically maintains a state which allows it to reason about the

monitored execution trace. Ideally, the amount of information needed to be stored by the monitor in its state depends only upon the property to be monitored and *not* upon the number of already processed events. This is desired because, due to the huge amount of events that can be generated during a monitoring session, one would want one's monitoring algorithms to work in linear time with the number of events processed.

There are, however, situations where it is not possible or practically feasible to use storage whose size is a function of only the monitoring requirement. One example is that of monitoring *extended regular expressions* (ERE), i.e., regular expressions enriched with a complement operator. As shown by the success of scripting languages like PERL or PYTHON, software developers tend to understand and like regular expressions and feel comfortable to describe patterns using those, so ERE is a good candidate formalism to specify monitoring requirements (we limit ourselves to only patterns described via temporal logics in this paper though).

It is however known that ERE to automata translation algorithms suffer from a non-elementary state explosion problem, because a complement operation requires nondeterministic-to-deterministic automata conversions, which yield exponential blowups in the number of states. Since complement operations can be nested, generating automata from EREs may often not be feasible in practice. Fortunately, there are algorithms which avoid this state explosion problem, at the expense of having to store the execution trace and then, at the end of the monitoring session, to analyze it by traversing it forwards and backwards many times. The interested reader is referred to [148] for a $O(n^3m)$ dynamic programming algorithm (n is the length of the execution trace and m is the size of the ERE), and to [269, 173] for $O(n^2m)$ non-trivial algorithms.

Based on these observations, we propose a first criterion to classify monitoring algorithms, namely on whether they *store or do not store the execution trace*. In the case of EREs, trace storing algorithms are polynomial in the size of the trace and linear in the ERE requirement, while the non-storing ones are linear in the size of the trace and highly exponential in the size of the requirement. In this paper we show that trace storing algorithms for linear temporal logic can be linear in both the trace and the requirement (see Section 8.2), while trace non-storing ones are linear in the size of the trace but simply exponential in the size of the requirement.

Trace non-storing algorithms are apparently preferred, but, however, their size can be so big that it could make their use unamenable in certain

important situations. One should carefully analyze the trade-offs in order to make the best choice in a particular situation.

1.1.2 Synchronous versus Asynchronous Monitoring

There are many safety critical applications in which one would want to report a violation of a requirement as soon as possible, and to not allow the monitored program to take any further action once a requirement is violated. We call this desired functionality *synchronous monitoring*. Otherwise, if a violation can only be detected after the monitored program executes several more steps or after it is stopped and its entire execution trace is needed to perform the analysis, then we call it *asynchronous monitoring*.

The dynamic programming algorithm presented in Section 8.2 is *not* synchronous, because one can detect a violation only after the program is stopped and its execution trace is available for backwards traversal. The algorithm presented in Section 8.3 is also asynchronous in general because there are universally false formulae which are detected so only at the end of an execution trace or only after several other execution steps. Consider, for example, that one monitors the finite trace LTL formula $\neg \langle \rangle (\Box A \vee \Box \neg A)$, which is false because at the end of any execution trace A either holds or not, or the formula $\Box \Box A \wedge \Box \Box \neg A$, which is also false but will be detected so only after two more events. However, the rewriting algorithm in Section 8.3 is synchronous in many practical situations. The algorithm in Section 8.4 is always synchronous, though one should be aware of the fact that its size may become a problem on large formulae.

In order for an LTL monitor to be synchronous, it needs to implement a validity checker for finite trace LTL, such as the one in Subsection 8.4.1 (Figure 8.2), and call it on the current formula after each event is processed. Checking validity of a finite trace LTL formula is very expensive (we are not aware of any theoretical result stating its exact complexity, but we believe that it is PSPACE-complete, like for standard infinite trace LTL [249]). We are currently considering providing a fully synchronous LTL monitoring module within JPAX, at the expense of calling a validity checker after each event, and let the user of the system choose either synchronous or asynchronous monitoring.

There are, however, many practical LTL formulae for which violation can be detected synchronously by the formula transforming rewriting-based algorithm presented in Section 8.3. Consider for example the sample formula of this paper, $\Box(\text{green} \rightarrow \neg \text{red} \vee \text{yellow})$, which is violated if and only

if a red event is observed after a green one. The monitoring requirement of our algorithm, which initially is the formula itself, will not be changed unless a green event is received, in which case it will change to $(\text{!red} \cup \text{yellow}) \wedge [](\text{green} \rightarrow \text{!red} \cup \text{yellow})$. A yellow event will turn it back into the initial formula, a green event will keep it unchanged, but a red event will turn it into `false`. If this is the case, then the monitor declares the formula violated and appropriate actions can be taken. Notice that the violation was detected *exactly* when it occurred. A very interesting, practical and challenging problem is to find criteria that say when a formula can be synchronously monitored without the use of a validity checker.

1.1.3 Predictive versus Exact Analysis

An increasingly important class of runtime analysis algorithms are concerned with *predicting* anomalies in programs from *successful* observed executions. One such algorithm can be easily obtained by slightly modifying the *wait-for-graph* algorithm, which is typically used to *detect* when a system is in a deadlock state, to make it predict deadlocks. One way to do this is to *not* remove synchronization objects from the wait-for-graph when threads/processes release them. Then even though a system is not deadlock, a warning can be reported to users if a cycle is found in the wait-for-graph, because that represents a *potential* of a deadlock.

Another algorithm falling into the same category is Eraser [233], a data race prediction procedure. For each shared memory region, Eraser maintains a set of *active locks* which protect it, which is intersected with the set of locks held by any accessing thread. If the set of active locks ever becomes empty then a warning is issued to the user, with the meaning that a potential unprotected access can take place. Both the deadlock and the data race predictive algorithms are very successful in practice because they scale well and find many of the errors they are designed for. We have also implemented improved versions of these algorithms in Java PathExplorer.

We are currently also investigating predictive analysis of safety properties expressed using past time temporal logic, and a prototype system called Java MultiPathExplorer is being implemented [240]. The main idea here is to *instrument* Java classes to emit events timestamped by vector clocks [99], thus enabling the observer to extract a *partial order* reflecting the causal dependency on the memory accesses of the multithreaded program. If any linearization of that inferred partial order leads to a violation of the safety property then a warning is generated to the user, with the meaning that

there can be executions of the multithreaded program, including the current one, which violate the requirements.

In this paper we restrict ourselves to only *exact* analysis of execution traces. That means that the events in the trace are supposed to have occurred exactly in the received order (this can be easily enforced by maintaining a logical clock, then timestamping each event with the current clock, and then delivering the messages in increasing order of timestamps), and that we only check whether that particular order violates the monitoring requirements or not. Techniques for predicting future time LTL violations will be investigated elsewhere soon.

Although the taxonomy discussed in this section is intended to only be applied to tools, the problem domain may also admit a similar taxonomy. While such a taxonomy seems to be hard to accomplish in general, it would certainly be very useful because it would allow one to choose the proper runtime analysis technique for a given system and set of properties. However, like this paper shows, it is often the case that one can choose among several types of runtime analysis techniques for a given problem domain.

Chapter 2

Background, Preliminaries, Notations

Add some structure to this chapter

from RV03 and RTA03

We let \mathbb{N} denote the set of natural numbers including 0 but excluding the infinity symbol ∞ and let \mathbb{N}_∞ denote the set $\mathbb{N} \cup \{\infty\}$. We also let \mathbb{Q} denote the set of rational numbers and \mathbb{R} the set of real numbers; as for natural numbers, the “ ∞ ” subscript can also be added to \mathbb{Q} and \mathbb{R} for the corresponding extensions of these sets. \mathbb{Q}^+ and \mathbb{R}^+ denote the sets of strictly positive (0 not included) rational and real numbers, respectively.

We fix a set Σ of elements called *events* or *states*. We call words in Σ^* *finite traces* and those in Σ^ω *infinite traces*. If $u \in \Sigma^* \cup \Sigma^\omega$ then u_i is the i -th state or event that appears in u . We call *finite-trace properties* sets $P \subseteq \Sigma^*$ of finite traces, *infinite-trace properties* sets $P \subseteq \Sigma^\omega$ of infinite traces, and just *properties* sets $P \subseteq \Sigma^* \cup \Sigma^\omega$ of finite or infinite traces. If the finite or infinite aspect of traces is understood from context, then we may call any of the types or properties above just *properties*. We may write $P(w)$ for a property P and a (finite or infinite) trace w whenever $w \in P$. Traces and properties are more commonly called *words* and *languages*, respectively, in the literature; we prefer to call them traces and properties to better reflect the intuition that our target application is monitoring and system observance, not formal languages. We take, however, the liberty to also call them words and languages whenever that terminology seems more appropriate.

In some cases states can be simply identified with their names, or labels, and specifications of properties on traces may just refer to those labels. For example, the regular expression $(s_1 \cdot s_2)^*$ specifies all those finite traces starting with state s_1 and in which states s_1 and s_2 alternate. In other cases, one can think of states as sets of atomic predicates, that is, predicates that hold in those states: if s is a state and a is an atomic predicate, then we say that $a(s)$ is true iff a “holds” in s ; thus, if all it matters with respect to states is which predicates hold and which do not hold in each state, then states can be faithfully identified with sets of predicates. We prefer to stay loose with respect to what “holds” means, because, depending on the context, it can mean anything. In conventional software situations, atomic predicates can be: boolean expressions over variables of the program, their satisfaction being decided by evaluating them in the current state of the program; or whether a function is being called or returned from; or whether a particular variable is being written to; or whether a particular lock is being held by a particular thread; and so on. In the presence of atomic predicates, specifications of properties on traces typically only refer to the atomic predicates. For example, the property “always a before b ”, that is, those traces containing no state in which b holds that is not preceded by some state in which a holds (for example, a can stand for “authentication” and b for “resource access”), can be expressed in LTL as the formula $\Box(b \rightarrow \Diamond a)$.

Let us recall some basic notions and notations from formal languages, temporarily using the consecrated terminology of “words” and “languages” instead of traces and properties. For an alphabet Σ , let \mathcal{L}_Σ be the set of languages over Σ , i.e., the powerset $\mathcal{P}(\Sigma^*)$. By abuse of language and notation, let \emptyset be the empty language $\{\}$ and ϵ the language containing only the empty word, $\{\epsilon\}$. If $L_1, L_2 \in \mathcal{L}_\Sigma$ then $L_1 \cdot L_2$ is the language $\{\alpha_1\alpha_2 \mid \alpha_1 \in L_1 \text{ and } \alpha_2 \in L_2\}$. Note that $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ and $L \cdot \epsilon = \epsilon \cdot L = L$. If $L \in \mathcal{L}_\Sigma$ then L^* is $\{\alpha_1\alpha_2 \cdots \alpha_n \mid n \geq 0 \text{ and } \alpha_1, \alpha_2, \dots, \alpha_n \in L\}$ and $\neg L$ is $\Sigma^* - L$.

We next recall some notions related to cardinality. If A is any set, we let $|A|$ denote the *cardinal* of A , which expresses the size of A . When A is finite, $|A|$ is precisely the number of elements of A and we call it a *finite cardinal*. Infinite sets can have different cardinals, called *transfinite* or even *infinite*. For example, natural numbers \mathbb{N} have the cardinal \aleph_0 (pronounced “aleph zero”) and real numbers \mathbb{R} have the cardinal c , also called the *cardinal of the continuum*. Two sets A and B are said to have the same cardinal, written $|A| = |B|$, iff there is some bijective mapping between the two. We write

$|A| \leq |B|$ iff there is some injective mapping from A to B .

The famous *Cantor-Bernstein-Schroeder theorem* states that if $|A| \leq |B|$ and $|B| \leq |A|$ then $|A| = |B|$. In other words, to show that there is some bijection between sets A and B , it suffices to find an injection from A to B and an injection from B to A . The two injections need not be bijections. For example, the inclusion of the interval $(0, 1)$ in \mathbb{R}^+ is obviously an injection, so $|(0, 1)| \leq |\mathbb{R}^+|$. On the other hand, the function $x \mapsto x/(2x + 1)$ from \mathbb{R}^+ to $(0, 1)$ (in fact its codomain is the interval $(0, 1/2)$) is also injective, so $|\mathbb{R}^+| \leq |(0, 1)|$. Neither of the two injective functions is bijective, yet by the Cantor-Bernstein-Schroeder theorem there is some bijection between $(0, 1)$ and \mathbb{R}^+ , that is, $|(0, 1)| = |\mathbb{R}^+|$. We will use this theorem to relate the various types of safety properties; for example, we will show that there is an injective function from safety properties over finite traces to safety properties over infinite traces and another injective function in the opposite direction. Unfortunately, the Cantor-Bernstein-Schroeder theorem is existential: it only says that some bijection exists between the two sets, but it does not give us an explicit bijection. Since the visualization of a concrete bijection between different sets of safety properties can be very meaningful, we will avoid using the Cantor-Bernstein-Schroeder theorem when we can find an explicit bijection between two sets of safety properties.

If A is a set of cardinal α , then 2^α is the cardinal of $\mathcal{P}(A)$, the power set of A (the set of subsets of A). It is known that $2^{\aleph_0} = c$, that is, there are as many sets of natural numbers as real numbers. The famous, still unanswered *continuum hypothesis*, states that there is no set whose size is strictly between \aleph_0 and c ; more generally, it states that, for any transfinite cardinal α , there is no proper cardinal between α and 2^α . If A and B are infinite sets, then $|A| + |B|$ and $|A| \cdot |B|$ are the cardinals of the sets $A \cup B$ and $A \times B$, respectively. An important property of transfinite cardinals is that of *absorption* – the larger cardinal absorbs the smaller one: if α and β are transfinite cardinals such that $\alpha \leq \beta$, then $\alpha + \beta = \alpha \cdot \beta = \beta$; in particular, $c \cdot 2^c = 2^c$. Besides sets of natural numbers, there are several other important sets that have cardinal c : streams (i.e., infinite sequences) of Booleans, streams of reals, non-empty closed or open intervals of reals, as well as the sets of all open or closed sets of reals, respectively (Exercise 1).

For our purposes, if Σ is an enumerable set of states, then Σ^* is also enumerable, so it has cardinal \aleph_0 . Also, if $|\Sigma| \leq c$, in particular if it is finite, then Σ^ω has the cardinal c , because it is equivalent to streams of states. We can then immediately infer that the set of finite-trace properties over Σ has

cardinal $2^{\aleph_0} = c$, while the set of infinite-trace properties has cardinal 2^c .

end from RV03 and RTA03

Exercises

Exercise 1 *Show that each of the following sets have cardinal c : streams (i.e., infinite sequences) of Booleans; streams of natural numbers; streams of real numbers; closed intervals of real numbers; open intervals of real numbers; closed sets of real numbers; open sets of real numbers.*

from J.ASE'05

2.1 Preliminaries

In this section we recall notions and notations that will be used in the paper, including membership equational logic, term rewriting, Maude notation, and (infinite trace) linear temporal logics.

2.1.1 Membership Equational Logic

Membership equational logic (MEL) extends many- and order-sorted equational logic by allowing memberships of terms to sorts in addition to the usual equational sentences. We only recall those MEL notions which are necessary for understanding this paper; the interested reader is referred to [201, 45] for a comprehensive exposition of MEL.

Basic Definition

A *many-kinded algebraic signature* (K, Σ) consists of a set K and a $(K^* \times K)$ -indexed set $\Sigma = \{\Sigma_{k_1 k_2 \dots k_n, k} \mid k_1, k_2, \dots, k_n, k \in K\}$ of operations, where an operation $\sigma \in \Sigma_{k_1 k_2 \dots k_n, k}$ is written $\sigma : k_1 k_2 \dots k_n \rightarrow k$. A *membership signature* Ω is a triple (K, Σ, π) where K is a set of *kinds*, Σ is a K -kinded algebraic signature, and $\pi : S \rightarrow K$ is a function that assigns to each element in its domain, called a *sort*, a kind. Therefore, sorts are grouped according to kinds and operations are defined on kinds. For simplicity, we will call a “membership signature” just a “signature” whenever there is no confusion.

For a *many-kinded signature* (K, Σ) , a Σ -algebra A consists of a K -indexed set $\{A_k \mid k \in K\}$ together with interpretations of operations $\sigma : k_1 k_2 \dots k_n \rightarrow k$ into functions $A_\sigma : A_{k_1} \times A_{k_2} \times \dots \times A_{k_n} \rightarrow A_k$. For any given signature $\Omega = (K, \Sigma, \pi)$, an Ω -membership algebra A is a Σ -algebra together with a set $A_s \subseteq A_{\pi(s)}$ for each sort $s \in S$. A particular algebra, called *term algebra*, is of special interest. Given a K -kinded signature Σ and a K -indexed set of *variables* X , let $T_\Sigma(X)$ be the algebra of Σ -terms over variables in X extending X iteratively as follows: if $\sigma : k_1 k_2 \dots k_n \rightarrow k$ and $t_1 \in T_{\Sigma, k_1}(X)$, $t_2 \in T_{\Sigma, k_2}(X)$, ..., $t_n \in T_{\Sigma, k_n}(X)$, then $\sigma(t_1, t_2, \dots, t_n) \in T_{\Sigma, k}(X)$.

Given a signature Ω and a K -indexed set of variables X , an *atomic* (Ω, X) -equation has the form $t = t'$, where $t, t' \in T_{\Sigma, k}(X)$, and an *atomic* (Ω, X) -membership has the form $t : s$, where s is a sort and $t \in T_{\Sigma, \pi(s)}(X)$. An Ω -sentence in MEL has the form $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, where a, a_1, \dots, a_n are atomic (Ω, X) -equations or (Ω, X) -memberships, and $\{a_1, \dots, a_n\}$ is a set (no duplications). If $n = 0$, then the Ω -sentence is called *unconditional* and written $(\forall X) a$. Equations are called *rewriting rules* when they are used only from left to right, as it will happen in this paper.

Given an Ω -algebra A and a K -kinded map $\theta : X \rightarrow A$, then $A, \theta \models_\Omega t = t'$ iff $\theta(t) = \theta(t')$, and $A, \theta \models_\Omega t : s$ iff $\theta(t) \in A_s$. A satisfies $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, written $A \models_\Omega (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, iff for each $\theta : X \rightarrow A$, if $A, \theta \models_\Omega a_1$ and ... and $A, \theta \models_\Omega a_n$, then $A, \theta \models_\Omega a$.

An Ω -specification (or Ω -theory) $T = (\Omega, E)$ in MEL consists of a signature Ω and a set E of Ω -sentences. An Ω -algebra A satisfies (or is a model of) $T = (\Omega, E)$, written $A \models T$, iff it satisfies each sentence in E .

Inference Rules

MEL admits complete deduction (see [201], where the rule of congruence is stated in a somewhat different but equivalent way). In the congruence rule below, $\sigma \in \Sigma_{k_1 \dots k_i, k}$, W is a set of variables $w_1 : k_1, \dots, w_{i-1} : k_{i-1}, w_{i+1} : k_{i+1}, \dots, w_n : k_n$, and $\sigma(W, t)$ is a shorthand for the term

$\sigma(w_1, \dots, w_{i-1}, t, w_{i+1}, \dots, w_n)$:

- (1) Reflexivity :
$$\frac{}{E \vdash_{\Omega} (\forall X) t = t}$$
- (2) Symmetry :
$$\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X) t' = t}$$
- (3) Transitivity :
$$\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t' = t''}{E \vdash_{\Omega} (\forall X) t = t''}$$
- (4) Congruence :
$$\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X, W) \sigma(W, t) = \sigma(W, t'), \text{ for each } \sigma \in \Sigma}$$
- (5) Membership :
$$\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t : s}{E \vdash_{\Omega} (\forall X) t' : s}$$
- (6) Modus Ponens :
$$\left\{ \begin{array}{l} \text{Given a sentence in } E \\ (\forall Y) t = t' \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m \\ \text{(resp. } (\forall Y) t : s \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m) \\ \text{and } \theta : Y \rightarrow T_{\Sigma}(X) \text{ s.t. for all } i \in \{1, \dots, n\} \text{ and } j \in \{1, \dots, m\} \\ E \vdash_{\Omega} (\forall X) \theta(t_i) = \theta(t'_i), E \vdash_{\Omega} (\forall X) \theta(w_j) : s_j \\ \hline E \vdash_{\Omega} (\forall X) \theta(t) = \theta(t') \quad (\text{resp. } E \vdash_{\Omega} (\forall X) \theta(t) : s) \end{array} \right.$$

The rules above can therefore prove any unconditional equation or membership that is true in all membership algebras satisfying E . In order to derive conditional statements, we will therefore consider the standard technique adapting the “deduction theorem” to equational logics, namely deriving the conclusion of the sentence after adding the condition as an axiom; in order for this procedure to be correct, the variables used in the conclusion need to be first transformed into constants. All variables can be transformed into constants, so we only consider the following simplified rules:

- (7) Theorem of Constants :
$$\frac{E \vdash_{\Omega \cup X} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}{E \vdash_{\Omega} (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n}$$
- (8) Implication Elimination :
$$\frac{E \cup \{a_1, \dots, a_n\} \vdash_{\Omega} (\forall \emptyset) a}{E \vdash_{\Omega} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}$$

Theorem 1 (from [201]) *With the notation above, $E \models_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$ if and only if $E \vdash_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$. Moreover, any statement can be proved by first applying rule (7), then (8), and then a series of rules (1) to (6).*

This theorem is used within the correctness proof of the monitoring algorithm in Section 8.3.

Initial Semantics and Induction

MEL specifications are often intended to allow only a restricted class of models (or algebras). For example, a specification of natural numbers defined using the Peano equational axioms would have many “undesired” models, such as models in which the addition operation is not commutative, or models in which, for example, $10=0$. We restrict the class of models of a MEL specification only to those which are *initial*, that is, those which obey the *no junk no confusion* principle; therefore, our specifications have *initial semantics* [116] in this paper. Intuitively, that means that only those models are allowed in which all elements can be “constructed” from smaller elements and in which no terms which cannot be proved equal are interpreted to the same elements.

By reducing the class of models, one can enlarge the class of sound inference rules. A major benefit one gets under initial semantics is that *proofs by induction become valid*. Since the proof of correctness for the main algorithm in this paper is done by induction on the structure of the temporal formula to monitor, it is important for the reader to be aware that the specifications presented from now on have initial semantics.

Syntactic Sugar Conventions

To make specifications easier to read, the following syntactic sugar conventions are widely accepted:

Subsorts. Given sorts s, s' with $\pi(s) = \pi(s') = k$, the declaration $s < s'$ is syntactic sugar for the conditional membership $(\forall x : k) x : s' \text{ if } x : s$.

Operations. If $\sigma \in \Omega_{k_1 \dots k_n, k}$ and $s_1, \dots, s_n, s \in S$ with $\pi(s_1) = k_1, \dots, \pi(s_n) = k_n, \pi(s) = k$, then the declaration $\sigma : s_1 \cdots s_n \rightarrow s$ is syntactic sugar for $(\forall x_1 : k_1, \dots, x_n : k_n) \sigma(x_1, \dots, x_n) : s \text{ if } x_1 : s_1 \wedge \cdots \wedge x_n : s_n$.

Variables. $(\forall x : s, X) \text{ a if } a_1 \wedge \dots \wedge a_n$ is syntactic sugar for the Ω -sentence $(\forall x : \pi(s), X) \text{ a if } a_1 \wedge \dots \wedge a_n \wedge x : s$. With this, the operation declaration $\sigma : s_1 \dots s_n \rightarrow s$ above is equivalent to $(\forall x_1 : s_1, \dots, x_n : s_n) \sigma(x_1, \dots, x_n) : s$.

2.1.2 Maude

Maude [72] is a freely distributed high-performance system in the OBJ [117] algebraic specification family, supporting both rewriting logic [200] and membership equational logic [201]. Because of its efficient rewriting engine, able to execute 3 million rewriting steps per second on standard PCs, and because of its metalanguage features, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, and even programming languages. We were delighted to notice how easily we could implement and efficiently validate our algorithms for testing LTL formulae on finite event traces in Maude, admittedly a tedious task in C++ or Java, and hence decided to use Maude at least for the prototyping stage of our runtime check algorithms.

We informally describe some of Maude's features via examples in this section, referring the interested reader to its manual [72] for more details. The examples discussed in this subsection are not random. On the one hand they show all the major features of Maude that we need, while on the other hand they are part of our current JPaX implementation; several references to them will be made later in the paper. Maude supports modularization in the OBJ style. There are various kinds of modules, but we use only functional modules which follow the pattern “`fmod <name> is <body> endfm`”, and which have initial semantics. The body of a functional module consists of a collection of declarations, of which we are using importation, sorts, subsorts, operations, variables and equations, usually in this order.

Defining Logics for Monitoring

In the following we introduce some modules that we think are general enough to be used within any logical environment for program monitoring that one would want to implement by rewriting. The first one simply defines atomic propositions as an abstract data type having one sort, `Atom`, and no operations or constraints:

```
fmod ATOM is
  sort Atom .
```

```
endfm
```

The actual names of atomic propositions will be automatically generated in another module that extends `ATOM`, as constants of sort `Atom`. These will be generated by the observer at the initialization of monitoring, from the actual properties that one wants to monitor.

An important concept in program monitoring is that of an (abstract) execution trace, which consists of a finite list of events. We abstract a single event by a list of atoms, those that hold after the action that generated the event took place. The values of the atomic propositions are updated by the observer according to the actual state of the executing program and then sent to Maude as a term of sort `Event` (more details regarding the communication between the running program and Maude will be given later):

```
fmod TRACE is
  protecting ATOM .
  sorts Event Event* Trace .
  subsorts Atom < Event < Event* < Trace .
  op empty : -> Event .
  op _ : Event Event -> Event [assoc comm id: empty prec 23] .
  var A : Atom .
  eq A A = A .
  op _* : Event -> Event* .
  op _,_ : Event Trace -> Trace [prec 25] .
endfm
```

The statement `protecting ATOM` imports the module `ATOM` without changing its initial semantics. The above is a compact way to use *mix-fix*¹ and order-sorted notation to define an abstract data type of traces: a trace is a comma separated list of events, where an event is itself a *set* of atoms. The `subsorts` declaration declares `Atom` to be a subsort of `Event`, which in turn is a subsort of `Event*` which is a subsort of `Trace`. Since elements of a subsort can occur as elements of a supersort without explicit lifting, we have as a consequence that a single event is also a trace, consisting of one event. Likewise, an atomic proposition can occur as an event, containing only this atomic proposition.

Operations can have attributes, such as associativity (A), commutativity (C), identity (I) as well as precedences, which are written between square brackets. When a binary operation is declared using the attributes A, C, and/or I, Maude uses built-in efficient specialized algorithms for matching

¹Underscores are places for arguments.

and rewriting. However, semantically speaking, the A, C, and/or I attributes can be replaced by there corresponding equations. The attribute `prec` gives a precedence to an operator², thus eliminating the need for most parentheses. Notice the special sort `Event*` which stays for terminal events, i.e., events that occur at the end of traces. Any event can potentially occur at the end of a trace. It is often the case that ending events are treated differently, like in the case of finite trace linear temporal logic; for this reason, we have introduced the operation `_*` which marks an event as terminal.

An event is defined as a set of atoms which should in fact be thought of as the set of all those atoms which “hold” in the new state of the event emitting program. Note the idempotency equation “`eq A A = A`”, which ensures that an event is indeed a set. On the other hand, a trace is a an ordered list of events which can potentially have repetitions of events. For example, the event “ $x = 5$ ” can occur several times during the execution of a program. Note that there is no need and consequently no definition of an empty trace.

Syntax and semantics are basic requirements to any logic. The following module introduces what we believe are the basic ingredients of monitoring logics, i.e., logics used for specifying monitoring requirements:

```
fmod LOGICS-BASIC is
  protecting TRACE .
  sort Formula .
  subsort Atom < Formula .
  ops true false : -> Formula .
  op [_] : Formula -> Bool .
  eq [true] = true .
  eq [false] = false .

  var A : Atom .
  var T : Trace .
  var E : Event .
  var E* : Event* .
  op _{ _ } : Formula Event* -> Formula [prec 10] .
  eq true{E*} = true .
  eq false{E*} = false .
  eq A{A E} = true .
  eq A{E} = false [owise] .
  eq A{E *} = A{E} .

  op _|=_ : Trace Formula -> Bool [prec 30] .
```

²The lower the precedence number, the tighter the binding.

```

eq T |= true  = true .
eq T |= false = false .
eq E  |= A = [A{E}] .
eq E,T |= A = E |= A .
endfm

```

The first block of declarations introduces the sort `Formula` which can be thought of as a generic sort for any well-formed formula in any logic. There are two designated formulae, namely `true` and `false`, with the obvious meaning in any monitoring logic. The sort `Bool` is built-in in Maude together with two constants `true` and `false`, which are different from those of sort `Formula`, and a generic operator `if_then_else_fi`. The “interpretation” operator `[_]` maps a formula to a Boolean value. Each logic implemented on top of LOGICS-BASIC is free to define it appropriately; here we only give the obvious mappings of `true` and `false` of `Formula` into `true` and `false` of `Bool`.

The second block defines the operation `_{}_` which takes a formula and an event and yields another formula. The intuition for this operation is that it “evaluates” the formula in the new state and produces a proof obligation as another formula for the subsequent events. If the returned formula is `true` or `false` then it means that the formula was satisfied or violated, regardless of the rest of the execution trace; in this case, a message can be returned by the observer. Each logic will further complete the definition of this operator. Note that the equation “`eq A{A E} = true`” speculates Maude’s capability of performing matching modulo associativity, commutativity and identity (the attributes of the *set* concatenation on events); it basically says that `A{E}` is `true` if `E` contains the atom `A`. The next equation contains the attribute `[owise]`, stating that it should be applied only if any other equation fails to apply at a particular position.

Finally, the satisfaction relation is defined. Two obvious equations deal with the formulae `true` and `false`. The last two equations state that a trace satisfies an atomic proposition `A` if evaluating that atomic proposition `A` on the first event in the trace yields `true`. The remaining elements in the trace do not matter because `A` is a simple atom, so it refers to only the current state.

Defining Propositional Calculus

A rewriting decision procedure for propositional calculus due to Hsiang [152] is adapted and presented. It provides the usual connectives `_&_` (and), `_+_` (exclusive or), `_\/_` (or), `!_` (negation), `_->_` (implication), and

$_<->_$ (equivalence). The procedure reduces tautological formulae to the constant `true` and all the others to some canonical form modulo associativity and commutativity. An unusual aspect of this procedure is that a canonical form consists of an exclusive or of conjunctions. In fact, this choice of basic operators corresponds to regarding propositional calculus as a Boolean ring rather than as a Boolean algebra. A major advantage of this choice is that normal forms are unique modulo associativity and commutativity. Even if propositional calculus is very basic to almost any logical environment, we decided to keep it as a separate logic instead of being part of the logic infrastructure of JPAX. One reason for this decision is that its operational semantics could be in conflict with other logics, for example ones in which conjunctive normal forms are desired.

An OBJ3 code for this procedure appeared in [117]. Below we give its obvious translation to Maude together with its finite trace semantics, noticing that Hsiang [152] showed that this rewriting system modulo associativity and commutativity is Church-Rosser and terminates. The Maude team was probably also inspired by this procedure, since the builtin `BOOL` module is very similar.

```
fmod PROP-CALC is
  extending LOGICS-BASIC .
*** Constructors ***
  op _/\_ : Formula Formula -> Formula [assoc comm prec 15] .
  op _++_ : Formula Formula -> Formula [assoc comm prec 17] .
  vars X Y Z : Formula .
  eq true /\ X = X .
  eq false /\ X = false .
  eq X /\ X = X .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .
*** Derived operators ***
  op _\/_ : Formula Formula -> Formula [assoc prec 19] .
  op !_ : Formula -> Formula [prec 13] .
  op _->_ : Formula Formula -> Formula [prec 21] .
  op _<->_ : Formula Formula -> Formula [prec 23] .
  eq X \/_ Y = X /\ Y ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ X /\ Y .
  eq X <-> Y = true ++ X ++ Y .
*** Finite trace semantics
  var T : Trace .
```

```

var E* : Event* .
eq T |= X /\ Y = T |= X and T |= Y .
eq T |= X ++ Y = T |= X xor T |= Y .
eq (X /\ Y){E*} = X{E*} /\ Y{E*} .
eq (X ++ Y){E*} = X{E*} ++ Y{E*} .
eq [X /\ Y] = [X] and [Y] .
eq [X ++ Y] = [X] xor [Y] .
endfm

```

The statement “`extending LOGICS-BASIC`” imports the module `LOGICS-BASIC` with the reserve that its initial semantics can be extended. The operators “`and`” and “`xor`” come from the Maude’s built-in module `BOOL` which is automatically imported by any other module.

Operators are declared with special attributes, such as `assoc` and `comm`, which enable Maude to use its specialized efficient internal rewriting algorithms. Once the module above is loaded³ in Maude, reductions can be done as follows:

```

reduce a -> b /\ c <-> (a -> b) /\ (a -> c) . ***> should be true
reduce a <-> ! b . ***> should be a ++ b

```

Notice that one should first declare the constants `a`, `b` and `c`. The last six equations in the module `PROP-CALC` are related to the semantics of propositional calculus. The default evaluation strategy for `[_]` is eager, so `[X]` will first evaluate `X` using propositional calculus reasoning and then will apply one of the last two equations if needed; these equations will not be applied normally in practical reductions, they are useful only in the correctness proof stated by Theorem 16.

end from J.ASE'05

³Either by typing it or using the command “`in <filename>`”.

Chapter 3

Safety Properties

safety property = every violation occurs after a finite execution.

later, when talking about LTL, discuss the classification of safety properties in [176]

In the literature, what we call “prefixes” are also called “good prefixes”, while the rest of the prefixes are called “bad prefixes”.

Intuitively, a safety property of a system is one stating that the system cannot “go wrong”, or, as Lamport [180] put it, that the “bad thing” never happens. In other words, in order for a system to violate a safety property, it should eventually “go wrong” or the “bad thing” should eventually happen. There is a very strong relationship between safety properties and runtime monitoring: if a safety property is violated by a running system, then the violation should happen *during* the execution of the system, in a finite amount of time, so a monitor for that property observing the running system should be able to detect the violation; an additional point in the favor of monitoring is that, if a system violates a safety property at some moment during its execution, then there is no way for the system to continue its execution to eventually satisfy the property, so a monitor needs not wait for a better future once it detects a bad present/past.

State properties or assertions that need only the current state of the running system to check whether they are violated or not, such as “no division by 0”, or “ x positive”, or no deadlock, are common safety properties;

once violated, one can stop the computation or take corrective measures. However, there are also interesting safety properties that involve more than one state of the system, such as “if one uses resource x then one must have authenticated at some moment in the past”, or “any start of a process must be followed by a stop within 10 units of time”, or “take command from user only if the user has logged in at some moment in the past and has not logged out since then”, etc. Needless to say that the atomic events, or states, which form execution traces on which safety properties are defined, can be quite abstract: not all the details of a system execution are relevant for the particular safety property of interest. In the context of monitoring, these relevant events or states can be extracted by means of appropriate instrumentation of the system. For example, runtime monitoring systems such as Tracematches [8] and MOP [56] use aspect-oriented technology to “hook” relevant observation points and appropriate event filters in a system.

It is customary to define safety properties as properties over *infinite traces*, to capture the intuition that they are defined for systems that can potentially run forever, such as reactive systems. A point in favor of infinite traces is that finite traces can be regarded as special cases of infinite traces, namely ones that “stutter” indefinitely in their last state (see, for example, Abadi and Lamport [4, 5]). Infinite traces are particularly desirable when one specifies safety properties using formalisms that have infinite-trace semantics, such as linear temporal logics or corresponding automata.

While “infinity” is a convenient abstraction that is relatively broadly-accepted nowadays in mathematics and in theoretical foundations of computer science, there is no evidence so far that a system can have an infinite-trace behavior (we have not seen any). A disclaimer is in place here: we do *not* advocate finite-traces as a foundation for safety properties; all we try to do is to argue that, just because they can be seen as a special case of infinite traces, finite traces are not entirely uninteresting. For example, a safety property associated to a one-time-access key issued to a client can be “activate, then use at most once, then close”. Using regular patterns over the alphabet of relevant events $\Sigma = \{activate, use, close\}$, this safety property can be expressed as “ $activate \cdot (\epsilon + use) \cdot close$ ”; any trace that is not a prefix of the language of this regular expression violates the property, including any other activation or use of the key after it was closed. While these finite-trace safety properties can easily be expressed as infinite-trace safety properties, we believe that that would be more artificial than simply accepting that in practice we deal with many finite-trace safety properties.

In this section we discuss various approaches to formalize safety properties and show that they are ultimately directly or indirectly equivalent. We categorize them into finite-trace safety properties, infinite-trace safety properties, and finite- and infinite-trace safety properties:

1. Section 3.1 defines safety properties over finite traces as prefix closed properties. A subset of finite-trace safety properties, that we call *persistent*, contain only traces that “have a future” within the property, that is, finite traces that can be continued into other finite traces that are also in the safety property. Persistent safety properties appear to be the right finite-trace variant that corresponds faithfully to the more conventional infinite-trace safety properties. Even though persistent safety properties form a proper subset of finite-trace safety properties and each finite-trace safety property has a largest persistent safety property included in it, we show that there is in fact a bijection between safety properties and persistent safety properties by showing them both to have the cardinal of the continuum c .
2. In Section 3.2, we consider two standard infinite-trace definitions of a safety property, one based on the intuition that violating behaviors must manifest so after a finite number of events and the other based on the intuition of a safety property as a closed set in an appropriate topology over infinite-traces. We show them both equivalent to persistent safety properties over finite traces, by constructing an explicit bijection (as opposed to using cardinality arguments and infer the existence of a bijection); consequently, infinite-trace safety properties also have the cardinal of the continuum c . Since closed sets of real numbers are in a bijective correspondence with the real numbers, we indirectly rediscover Alpern and Schneider’s result [11] stating that infinite-trace safety properties correspond to closed sets in infinite-trace topology.
3. Section 3.3 considers safety properties defined over both finite and infinite traces. We discuss two definitions of such safety properties encountered in the literature, and, using cardinality arguments, we show their equivalence with safety properties over only finite traces. In particular, safety properties over finite and infinite traces also have the cardinality of the continuum c . We also show that prefix-closeness is not a sufficient condition to characterize (not even bijectively) such safety properties, by showing that there are significantly more (2^c) prefix-closed properties over finite and infinite traces than safety properties.

Therefore, each of the classes of safety properties is in bijection with the real numbers. Since there are so many safety properties, we can also insightfully conclude that there is *no* enumerable mechanism to define all the safety properties, because $\aleph_0 \leq c$. Therefore, particular logical or syntactic recursive formalisms can only define *some* of the safety properties, but not all of them.

3.1 Finite Traces

One of the most common intuitions for a safety property is as a prefix-closed set of finite traces. This captures best the intuition that once something bad happened, there is no way to recover: if $w \notin P$ then there is no u such that $P(wu)$, which is equivalent to saying that if $P(wu)$ then $P(w)$, which is equivalent to saying that P is prefix closed. From a monitoring perspective, a prefix closed property can be regarded as one containing all the good (complete or partial) behaviors of the observed system: once a state is encountered that does not form a good behavior together with the previously observed states, then a violation can be reported.

Definition 1 *Let $\text{prefixes} : \Sigma^* \rightarrow \mathcal{P}(\Sigma^*)$ be the prefix function returning for any finite trace all its prefixes, and let $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ be its corresponding closure operator that takes sets of finite traces and closes them under prefixes.*

Note that $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ is indeed a closure operator (Exercise 2): it is extensive ($P \subseteq \text{prefixes}(P)$), monotone ($P \subseteq P'$ implies $\text{prefixes}(P) \subseteq \text{prefixes}(P')$), and idempotent ($\text{prefixes}(\text{prefixes}(P)) = \text{prefixes}(P)$).

Definition 2 *Let Safety^* be the set of finite-trace prefix-closed properties, that is, the set $\{P \in \mathcal{P}(\Sigma^*) \mid P = \text{prefixes}(P)\}$. In other words, Safety^* is the set of fixed points of the prefix operator $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$.*

The star superscript in Safety^* reflects that its traces are finite; in the next section we will define a set Safety^ω of infinite-trace safety properties. Since $\text{prefixes}(P) \in \text{Safety}^*$ for any $P \in \mathcal{P}(\Sigma^*)$, we can assume from here on that $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ is actually a function $\mathcal{P}(\Sigma^*) \rightarrow \text{Safety}^*$.

Example 1 Consider the one-time-access key safety property discussed above, saying that a client can “activate, then use at most once, and then

close” the key. If $\Sigma = \{\text{activate}, \text{use}, \text{close}\}$, then this safety property can be expressed as the finite set of finite words

$$\{\epsilon, \text{activate}, \text{activate close}, \text{activate use}, \text{activate use close}\}$$

No other behavior is allowed. Now suppose that the safety policy is extended to allow multiple uses of the key once activated, but still no further events once it is closed. The extended safety property has now infinitely many finite-traces:

$$\{\epsilon\} \cup \{\text{activate}\} \cdot \{\text{use}^n \mid n \in \mathbb{N}\} \cdot \{\epsilon, \text{close}\}.$$

Note that this property is indeed prefix-closed. A monitor in charge of online checking this safety property would report a violation if the first event is not *activate*, or if it encounters any second *activate* event, or if it encounters any event after a *close* event is observed, including another *close* event.

It is interesting to note that this finite-trace safety property encompasses both finite and infinite aspects. For example, it does not preclude behaviors in which one sees an *activate* event and then an arbitrary number of *use* events; *use* events can persist indefinitely after an *activate* event without violating the property. On the other hand, once a *close* event is encountered, no other event can be further seen. We will shortly see that the safety property above properly includes the *persistent* safety property $\{\epsilon\} \cup \{\text{activate use}^n \mid n \in \mathbb{N}\}$, which corresponds to the infinite-trace safety property $\{\text{activate use}^\omega\}$. \square

While prefix closeness seems to be the right requirement for a safety property, one can argue that it is not sufficient. For example, in the context of reactive systems that supposedly run forever, one may think of a safety property as one containing safe finite traces, that is, ones for which the reactive system can always find a way to continue its execution safely. The definition of safety properties above includes, among other safety properties, the empty set of traces as well as all prefix-closed *finite* sets of finite traces; any reactive system will eventually violate such safety properties, so one can say that the definition of safety property above is too generous.

We next define *persistent safety properties* as ones that always allow a future; intuitively, an observed reactive system that is in a safe state can always (if persistent enough) find a way to continue its execution to a next safe state. This notion is reminiscent of “feasibility”, a semantic characterization of fairness in [15], and of “machine closeness” [4, 234], also used in the context of fairness.

Definition 3 Let $\text{PersistentSafety}^*$ be the set of finite-trace persistent safety properties, that is, safety properties $P \in \text{Safety}^*$ such that if $P(w)$ for some $w \in \Sigma^*$ then there is some $a \in \Sigma$ such that $P(wa)$.

If a persistent safety property is non-empty, then note that it must contain an infinite number of words. The persistency aspect of a finite-trace safety property can be regarded, in some sense, as a liveness argument. Indeed, assuming that it is a “good thing” for a trace to be indefinitely continued, then a persistent safety property is one in which the “good thing” always eventually happens. If one takes the liberty to regard “stuck” computations as unfair, then the persistency aspect above can also be regarded as a fairness argument.

Another way to think of persistent safety properties is as a means to refer to infinite behaviors by means of finite traces. This view is, in some sense, dual to the more common approach to regard finite behaviors as infinite behaviors that stutter infinitely in a “last” state (see, for example, Abadi and Lamport [4, 5] for a formalization of such last-state infinite stuttering).

Note that if Σ is a degenerate set of events containing only one element, that is, if $|\Sigma| = 1$, then $|\text{Safety}^*| = \aleph_0$ and $|\text{PersistentSafety}^*| = 2$; indeed, if $\Sigma = \{a\}$ then Safety^* contains precisely the finite properties $a^{\leq n} = \{a^i \mid 0 \leq i \leq n\}$ for each $n \in \mathbb{N}$ plus the infinite property $\{a^n \mid n \in \mathbb{N}\}$, so a total of $\aleph_0 + 1 = \aleph_0$ properties, while $\text{PersistentSafety}^*$ contains only two properties, namely \emptyset and $\{a^n \mid n \in \mathbb{N}\}$. The case when there is only one event or state in Σ is neither interesting nor practical. Therefore, from here on in this paper we take the liberty to assume that $|\Sigma| \geq 2$. Since in practice Σ contains states or events generated by a computer, for simplicity in stating some of the subsequent results, we also take the liberty to assume that $|\Sigma| \leq \aleph_0$; therefore, Σ can be any finite or recursively enumerable set, including \mathbb{N} , \mathbb{N}_∞ , \mathbb{Q} , etc., but cannot be \mathbb{R} or any set “larger” than \mathbb{R} . With these assumptions, it follows that $|\Sigma^*| = \aleph_0$ (finite words are recursively enumerable) and $|\Sigma^\omega| = c$ (infinite streams have the cardinality of the continuum).

Proposition 1 Safety^* and $\text{PersistentSafety}^*$ are closed under union; Safety^* is also closed under intersection.

Proof: The union and the intersection of prefix-closed properties is also prefix-closed. Also, the union of persistent prefix-closed properties is also persistent. \square

The intersection of persistent safety properties may not be persistent:

Example 2 Let Σ be the set $\{0, 1\}$. Let $P = \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\}$ be two persistent safety properties, where ϵ is the empty word (the word containing no letters). Then $P \cap P'$ is the finite safety property $\{\epsilon, 1\}$, which is not persistent. If one thinks that this happened because $P \cap P'$ does not contain any proper (i.e., non-empty) persistent property, then one can take instead the persistent safety properties $P = \{0^n \mid n \in \mathbb{N}\} \cdot \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{0^n \mid n \in \mathbb{N}\} \cdot (\{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\})$, whose intersection is the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. This safety property is not persistent because its words ending in 1 cannot persist, but it contains the proper persistent safety property $\{0^n \mid n \in \mathbb{N}\}$. \square

Therefore, we can associate to any safety property in Safety^* a largest persistent safety property in $\text{PersistentSafety}^*$, by simply taking the union of all persistent safety properties that are included in the original safety property (the empty property is one of them, the smallest):

Definition 4 For a safety property $P \in \text{Safety}^*$, let $P^\circ \in \text{PersistentSafety}^*$ be the largest persistent safety property with $P^\circ \subseteq P$.

The following example shows that one may need to eliminate infinitely many words from a safety property in order to obtain a persistent safety property:

Example 3 Let $\Sigma = \{0, 1\}$ and let P be the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. Then P° can contain no word ending with a 1 and can contain all the words of 0's. Therefore, $P^\circ = \{0^n \mid n \in \mathbb{N}\}$. \square

Finite safety properties obviously cannot contain any non-empty persistent safety property, that is, $P^\circ = \emptyset$ if P is finite. But what if P is infinite? Is it always the case that it contains a non-empty persistent safety property? Interestingly, it turns out that this is true if and only if Σ is finite:

Proposition 2 If Σ is finite and P is a safety property containing infinitely many words, then $P^\circ \neq \emptyset$.

Proof: For each letter $a \in \Sigma$, let us define the *derivative of P wrt a* , written $\delta_a(P)$, as the language $\{w \in \Sigma^* \mid aw \in P\}$. Since

$$P = \{\epsilon\} \cup \bigcup_{a \in \Sigma} \{a\} \cdot \delta_a(P)$$

since Σ is finite, and since P is infinite, it follows that there is some $a_1 \in \Sigma$ such that $\delta_{a_1}(P)$ is infinite; note that $a_1 \in P$ since P is prefix closed. Similarly, since $\delta_{a_1}(P)$ is infinite, there is some $a_2 \in \Sigma$ such that $\delta_{a_2}(\delta_{a_1}(P))$ is infinite and $a_1 a_2 \in P$. Iterating this reasoning, we can find some $a_n \in \Sigma$ for each $n \in \mathbb{N}$, such that $a_1 a_2 \dots a_n \in P$ and $\delta_{a_n}(\dots(\delta_{a_2}(\delta_{a_1}(P)))\dots)$ is infinite, that is, the set $\{w \in \Sigma^* \mid a_1 a_2 \dots a_n w \in P\}$ is infinite. It is now easy to see that the set $\{a_1 a_2 \dots a_n \mid n \in \mathbb{N}\} \subseteq P$ is persistent. Therefore, $P^\circ \neq \emptyset$. \square

The following example shows that Σ must indeed be finite in order for the result above to hold:

Example 4 Consider some infinite set of events or states Σ . Then we can label distinct elements in Σ with distinct labels in $\mathbb{N} \cup \{\infty\}$. We only need these elements from Σ ; therefore, without loss of generality, we can assume that $\Sigma = \mathbb{N} \cup \{\infty\}$. Let P be the safety property

$$\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid 0 \leq m \leq n+1\},$$

where ϵ is the empty word (the word containing no letters) and $n \dots (n+1)$ is also the empty word for any $n \in \mathbb{N}$. Then P° is the empty property. Indeed, note that any persistent safety property P' included in P cannot have traces ending in 0, because those cannot be continued into other traces in P ; since P' cannot contain traces ending in 0, it cannot contain traces ending in 1 either, because such traces can only be continued with a 0 letter into traces in P , but those traces have already been decided that cannot be part of P' ; inductively, one can show that P' can contain no words ending in letters that are natural numbers in \mathbb{N} . Since the only trace in P ending in ∞ is ∞ itself and since ∞ can only be continued with a natural number letter into a trace in P but such trace cannot belong to P' , we deduce that P' can contain no word with letters in Σ . In particular, P° must be empty. \square

Even though we know that the largest persistent safety property P° included into a safety property P always exists because **PersistentSafety**^{*} is closed under union, we would like to have a more constructive way to obtain it. A first and obvious thing to do is to eliminate from P all the “stuck” computations, that is, those which cannot be added any new state to obtain a trace that is also in P . This removal step does not destroy the prefix-closeness of P , but it may reveal new computations which are stuck. By iteratively eliminating all the computations that get stuck in a finite number of steps, one would expect to obtain a persistent safety property,

namely precisely P° . It turns out that this is indeed true only if Σ is finite. If that is the case, then the following can also be used as an alternative definition of P° :

Proposition 3 *Given safety property $P \in \mathbf{Safety}^*$, then let P^- be the property $\{w \in P \mid (\exists a \in \Sigma) wa \in P\}$. Also, let $\{P_i \mid i \in \mathbb{N}\}$ be properties defined as $P_0 = P$ and $P_{i+1} = P_i^-$ for all $i \geq 0$. Then $P^\circ = \bigcap_{i \geq 0} P_i$ whenever Σ is finite.*

Proof: It is easy to see that if P is prefix-closed then $P^- \subseteq P$ is also prefix-closed, so P^- is also a property in \mathbf{Safety}^* . Therefore, the properties P_i form a sequence $P = P_0 \supseteq P_1 \supseteq P_2 \supseteq \dots$ of increasingly smaller safety properties.

Let us first prove that $\bigcap_{i \geq 0} P_i$ is a persistent safety property. Assume by contradiction that for some $w \in \bigcap_{i \geq 0} P_i$ there is no $a \in \Sigma$ such that $wa \in \bigcap_{i \geq 0} P_i$. In other words, we can find for each $a \in \Sigma$ some $i_a \geq 0$ such that $wa \notin P_{i_a}$. Since Σ is finite, we can let i be the largest among the natural numbers $i_a \in \mathbb{N}$ for all $a \in \Sigma$. Since $P_i \subseteq P_{i_a}$ for all $a \in \Sigma$, it should be clear that there is no $a \in \Sigma$ such that $wa \in P_i$, which means that $w \notin P_{i+1}$. This contradicts the fact that $w \in \bigcap_{i \geq 0} P_i$. Therefore, $\bigcap_{i \geq 0} P_i \in \mathbf{PersistentSafety}^*$.

Let us now prove that $\bigcap_{i \geq 0} P_i$ is the largest persistent safety property included in P . Let P' be any persistent safety property included in P . We show by induction on i that $P' \subseteq P_i$ for all $i \in \mathbb{N}$. The base case, $P' \subseteq P_0$, is obvious. Suppose that $P' \subseteq P_i$ for some $i \in \mathbb{N}$ and let $w \in P'$. Since P' is persistent, there is some $a \in \Sigma$ such that $wa \in P' \subseteq P_i$, which means that $w \in P_{i+1}$. Since w was chosen arbitrarily, it follows that $P' \subseteq P_{i+1}$. Therefore, $P' \subseteq \bigcap_{i \geq 0} P_i$. \square

We next show that the finiteness of Σ was a necessary requirement in order for the result above to hold. In other words, we show that if Σ is allowed to be infinite then we can find a safety property $P \in \mathbf{Safety}^*$ over Σ such that $P^\circ \in \mathbf{PersistentSafety}^*$ and $\bigcap_{i \geq 0} P_i \in \mathbf{Safety}^*$ are distinct. Since we showed in the proof of Proposition 3 that any persistent safety property P' is included in $\bigcap_{i \geq 0} P_i$, it follows that $P^\circ \subseteq \bigcap_{i \geq 0} P_i$. Since P° is the largest persistent safety property included in P , one can easily show that $P^\circ = (\bigcap_{i \geq 0} P_i)^\circ$. Therefore, it suffices to find a safety property P such that $\bigcap_{i \geq 0} P_i$ is not persistent, which is what we do in the next example:

Example 5 Consider the safety property P over infinite $\Sigma = \mathbb{N} \cup \{\infty\}$ discussed in Example 4, namely $\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid 0 \leq m \leq$

$n + 1\}$. Then one can easily show by induction on $i \in \mathbb{N}$ that the properties P_i defined in Proposition 3 are the sets $\{\epsilon\} \cup \{\infty n(n-1) \dots (m+1)m \mid i \leq m \leq n+1\}$; in other words, each P_i excludes from P all the words whose last letters are smaller than i when regarded as natural numbers. Then the intersection $\bigcap_{i \geq 0} P_i$ contains no trace ending in a natural number; the only possibility left is then $\bigcap_{i \geq 0} P_i = \{\epsilon, \infty\}$, which is different from $P^\circ = \emptyset$ (see Example 4).

One may argue that $P^\circ \neq \bigcap_{i \geq 0} P_i$ above happened precisely because P° was empty. One can instead pick the safety property $Q = \{0^n \mid n \in \mathbb{N}\} \cdot P$. Then one can show following the same idea as in Example 4 that $Q^\circ = \{0^n \mid n \in \mathbb{N}\}$. Further, one can show that $Q_i = \{0^n \mid n \in \mathbb{N}\} \cdot P_i$, so $\bigcap_{i \geq 0} Q_i = \{0^n \mid n \in \mathbb{N}\} \cup \{0^n \infty \mid n \in \mathbb{N}\}$, which is different from Q° . \square

Persistency is reminiscent of “feasibility” introduced by Apt et al. [15] in the context of fairness, and of “machine closeness” introduced by Abadi and Lamport [4, 5] (see also Schneider [234]) in the context of refinement. Let us use the terminology “machine closeness”: a property L (typically a liveness or a fairness property) is *machine closed* for a property M (typically given as the language of some state machine) iff L does not prohibit any of the observable runtime behaviors of M , that is, iff $\text{prefixes}(M) = \text{prefixes}(M \cap L)$; for example, if M is the total property (i.e., every event is possible at any moment, i.e., $M = \Sigma^*$) and L is the property stating that “always eventually event a ”, then any prefix of M can be continued to obtain a property satisfying L . Persistency is related to machine closeness in that a safety property P is persistent if and only if P° is machine closed for P . In other words, there is nothing P can do in a finite amount of time that P° cannot do. However, there is a caveat here: since liveness and fairness are inherently infinite-trace notions, machine closeness (or feasibility) have been introduced in the context of infinite-traces. On the other hand, persistency makes sense only in the context of finite traces.

It is clear that $\text{PersistentSafety}^*$ is properly included in Safety^* . Yet, we next show that, surprisingly, there is a bijective correspondence between Safety^* and $\text{PersistentSafety}^*$, both having the cardinal of the continuum:

Theorem 2 $|\text{PersistentSafety}^*| = |\text{Safety}^*| = c$.

Proof: Since Σ^* is recursively enumerable and since $2^{\aleph_0} = c$, we can readily infer that $|\text{PersistentSafety}^*| \leq |\text{Safety}^*| \leq |\mathcal{P}(\Sigma^*)| = c$.

Let us now define an injective function φ from the open interval of real numbers $(0, 1)$ to $\text{PersistentSafety}^*$. Since $|\Sigma| \geq 2$, let us distinguish

two different elements in Σ and let us label them $\bar{0}$ and $\bar{1}$. For a real $r \in (0, 1)$, let $\varphi(r)$ be the set $\{\bar{\alpha} \mid \alpha \in \{0, 1\}^* \text{ and } 0.\alpha < r\}$, where $0.\alpha$ is the (rational) number in $(0, 1)$ whose decimals in binary representation are α , and where $\bar{\alpha}$ is the word in Σ^* corresponding to α . Note that the set $\varphi(r) \in \mathcal{P}(\Sigma^*)$ is prefix-closed for any $r \in (0, 1)$, and that if $w \in \varphi(r)$ then also $w\bar{0} \in \varphi(r)$ (the latter holds since, by real numbers conventions, $0.\alpha = 0.\alpha 0$), so $\varphi(r) \in \text{PersistentSafety}^*$. Since the set of rationals with finite number of decimals in binary representation is dense in \mathbb{R} (i.e., it intersects any open interval in \mathbb{R}) and in particular in the interval $(0, 1)$, it follows that the function $\varphi : (0, 1) \rightarrow \text{PersistentSafety}^*$ is injective: indeed, if $r_1 \neq r_2 \in (0, 1)$, say $r_1 < r_2$, then there is some $\alpha \in \{0, 1\}^*$ such that $r_1 < 0.\alpha < r_2$, so $\varphi(r_1) \neq \varphi(r_2)$. Since the interval $(0, 1)$ has the cardinal of the continuum c , the existence of the injective function φ implies that $c \leq |\text{PersistentSafety}^*|$. By the Cantor-Bernstein-Schroeder theorem it follows that $|\text{PersistentSafety}^*| = |\text{Safety}^*| = c$. \square

From safety: *The proof above could have been rearranged to avoid the need to use the set $\text{PersistentSafety}^*$. However, we prefer to keep it for two reasons:*

1. *For finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix closed properties;*
 2. *Persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c .*
-

With regards to finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix-closed properties. Also, persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c .

3.2 Infinite Traces

The finite-trace safety properties defined above, persistent or not, rely on the intuition of a correct prefix: a safety property is identified with the set of all its finite prefixes. In the case of a persistent safety property, each “informal”

infinite acceptable behavior is captured by its infinite set of finite prefixes. Even though persistent safety properties appear to capture well in a finite-trace setting the intuition of safety in the context of (infinite-trace) reactive systems, one could argue that it does not say anything about unacceptable infinite traces. Indeed, one may think that persistent safety properties do not capture the intuition that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable. In this section we show that there is in fact a bijection between safety properties over infinite traces and persistent safety properties over finite traces as we defined them in the previous section.

We start by extending the `prefixes` function to infinite traces:

Definition 5 *Let $\text{prefixes} : \Sigma^\omega \rightarrow \mathcal{P}(\Sigma^*)$ be the function returning for any infinite trace u all its finite prefixes $\text{prefixes}(u)$, and let $\text{prefixes} : \mathcal{P}(\Sigma^\omega) \rightarrow \mathcal{P}(\Sigma^*)$ be its corresponding extension to sets of infinite traces.*

Note that $\text{prefixes}(S) \in \text{PersistentSafety}^*$ for any $S \in \mathcal{P}(\Sigma^\omega)$, so `prefixes` is in fact a function $\mathcal{P}(\Sigma^\omega) \rightarrow \text{PersistentSafety}^*$.

The definition of safety properties over infinite traces below appears to be the most used definition of a safety property in the literature; at our knowledge, it was formally introduced by Alpern and Schneider [11], but they credit the insights of their definition to Lamport [180].

Definition 6 *Let Safety^ω be the set of infinite-trace properties $Q \in \mathcal{P}(\Sigma^\omega)$ s.t.: if $u \notin Q$ then there is a finite trace $w \in \text{prefixes}(u)$ s.t. $wv \notin Q$ for any $v \in \Sigma^\omega$.*

In other words, if an infinite behavior violates the safety property then there is some finite-trace “violation threshold”; once the violation threshold is reached, there is no chance to recover.

The following proposition can serve as an alternative and more compact definition of Safety^ω :

Proposition 4 $\text{Safety}^\omega = \{Q \in \mathcal{P}(\Sigma^\omega) \mid u \in Q \text{ iff } \text{prefixes}(u) \subseteq \text{prefixes}(Q)\}.$

Proof: Since $u \in Q$ implies $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$, the only thing left to show is that $Q \in \text{Safety}^\omega$ iff “ $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ implies $u \in Q$ ”; the latter is equivalent to “ $u \notin Q$ implies $\text{prefixes}(u) \not\subseteq \text{prefixes}(Q)$ ”, which is further equivalent to “ $u \notin Q$ implies there is some $w \in \text{prefixes}(u)$ s.t. $w \notin \text{prefixes}(Q)$ ”, which is indeed equivalent to $Q \in \text{Safety}^\omega$. \square

Another common intuition for safety properties over infinite traces is as *closed* sets in the topology corresponding to Σ^ω . Alpern and Schneider captured formally this intuition for the first time in [11]; then it was used as a convenient definition of safety by Abadi and Lamport [4, 5] among others:

Definition 7 *An infinite sequence $u^{(1)}, u^{(2)}, \dots$, of infinite traces in Σ^ω converges to $u \in \Sigma^\omega$, or u is a limit of $u^{(1)}, u^{(2)}, \dots$, written $u = \lim_i u^{(i)}$, iff for all $m \geq 0$ there is an $n \geq 0$ such that $u_1^{(i)} u_2^{(i)} \dots u_m^{(i)} = u_1 u_2 \dots u_m$ for all $i \geq n$. If $Q \in \mathcal{P}(\Sigma^\omega)$ then \overline{Q} , the closure of Q , is the set $\{\lim_i u^{(i)} \mid u^{(i)} \in Q \text{ for all } i \in \mathbb{N}\}$.*

It can be easily shown that the overline closure above is indeed a closure operator on Σ^ω , that is, it is extensive ($Q \subseteq \overline{Q}$), monotone ($Q \subseteq Q'$ implies $\overline{Q} \subseteq \overline{Q'}$), and idempotent ($\overline{\overline{Q}} = \overline{Q}$); see Exercise 6.

Definition 8 *Let $\text{Safety}_{\lim}^\omega$ be the set of properties $\{Q \in \mathcal{P}(\Sigma^\omega) \mid Q = \overline{Q}\}$.*

As expected, the two infinite-trace safety property definitions are equivalent; we have not found any formal proof in the literature, so for the sake of completeness we give a simple proof here:

Proposition 5 $\text{Safety}_{\lim}^\omega = \text{Safety}^\omega$.

Proof: All we need to prove is that for any $Q \in \mathcal{P}(\Sigma^\omega)$ and any $u \in \Sigma^\omega$, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ iff $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}, u^{(2)}, \dots$ in Q . If $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ then one can find for each $i \geq 0$ some $u^{(i)} \in Q$ such that $u_1 u_2 \dots u_i = u_1^{(i)} u_2^{(i)} \dots u_i^{(i)}$, so for each $m \geq 0$ one can pick $n = m$ such that $u_1 u_2 \dots u_m = u_1^{(i)} u_2^{(i)} \dots u_m^{(i)}$ for all $i \geq n$, so $u = \lim_i u^{(i)}$. Conversely, if $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}, u^{(2)}, \dots$ in Q , then for any $m \geq 0$ there is some $n \geq 0$ such that $u_1 u_2 \dots u_m = u_1^{(n)} u_2^{(n)} \dots u_m^{(n)}$, that is, for any prefix of u there is some $u' \in Q$ having the same prefix, that is, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. \square

The next result establishes the relationship between infinite-trace safety properties and finite-trace persistent safety properties, by proposing a concrete bijective mapping relating the two (as opposed to using cardinality arguments to indirectly show only the existence of such a mapping). Therefore, there is also a bijective correspondence between safety properties over infinite traces and the real numbers:

Note there are 2^c properties over Σ^ω

Theorem 3 $|\text{Safety}^\omega| = |\text{PersistentSafety}^*| = c$.

Proof: We show that there is a bijective function between the two sets of safety properties. Recall that $\text{prefixes}(S) \in \text{PersistentSafety}^*$ for any $S \in \mathcal{P}(\Sigma^\omega)$, that is, that prefixes is a function $\mathcal{P}(\Sigma^\omega) \rightarrow \text{PersistentSafety}^*$. Let $\text{prefixes} : \text{Safety}^\omega \rightarrow \text{PersistentSafety}^*$ be the restriction of this prefix function to Safety^ω . Let us also define a function $\omega : \text{PersistentSafety}^* \rightarrow \text{Safety}^\omega$ as follows: $\omega(P) = \{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$. This function is well-defined: if $u \notin \omega(P)$ then by the definition of $\omega(P)$ there is some $w \in \text{prefixes}(u)$ such that $w \notin P$; since $w \in \text{prefixes}(wv)$ for any $v \in \Sigma^\omega$, it follows that $wv \notin \omega(P)$ for any $v \in \Sigma^\omega$.

We next show that prefixes and ω are inverse to each other. Let us first show that $\text{prefixes}(\omega(P)) = P$ for any $P \in \text{PersistentSafety}^*$. The inclusion $\text{prefixes}(\omega(P)) \subseteq P$ follows by the definition of $\omega(P)$: $\text{prefixes}(u) \subseteq P$ for any $u \in \omega(P)$. The inclusion $P \subseteq \text{prefixes}(\omega(P))$ follows from the fact that P is a persistent safety property: for any $w \in P$ one can iteratively build an infinite sequence v_1, v_2, \dots , such that $wv_1, wv_1v_2, \dots \in P$, so $wv_1v_2\dots \in \omega(P)$. Let us now show that $\omega(\text{prefixes}(Q)) = Q$ for any $Q \in \text{Safety}^\omega$. The inclusion $Q \subseteq \omega(\text{prefixes}(Q))$ is immediate. For the other inclusion, let $u \in \omega(\text{prefixes}(Q))$, that is, $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. Suppose by contradiction that $u \notin Q$. Then there is some $w \in \text{prefixes}(u)$ such that $wv \notin Q$ for any $v \in \Sigma^\omega$. Since $w \in \text{prefixes}(u)$ and $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$, it follows that $w \in \text{prefixes}(Q)$, that is, that there is some $u' \in Q$ such that $u' = wv$ for some $v \in \Sigma^\omega$. This contradicts the fact that $wv \notin Q$ for any $v \in \Sigma^\omega$. Consequently, $u \in Q$.

The second part follows by Theorem 2. \square

3.3 Finite and Infinite Traces

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [181] extends the notion of safety in Definition 6 to properties over both finite and infinite traces, while Schneider et al [235, 123] give an alternative definition of safety over finite and infinite traces. We define both approaches shortly and then show their equivalence and their bijective correspondence with real numbers. Before that, we argue that the mix of finite and infinite traces is less trivial than it may appear,

by showing that there are significantly more prefix closed properties than in the case when only finite traces were considered.

Definition 9 Let $\text{PrefixClosed}^{*,\omega}$ be the set of prefix-closed sets of finite and infinite traces: for $Q \subseteq \Sigma^* \cup \Sigma^\omega$, $Q \in \text{PrefixClosed}^{*,\omega}$ iff $\text{prefixes}(Q) \subseteq Q$. Also, let $\text{PersistentPrefixClosed}^{*,\omega}$ be the set of persistent prefix-closed sets of finite and infinite traces: for $Q \in \text{PrefixClosed}^{*,\omega}$, it is the case that $Q \in \text{PersistentPrefixClosed}^{*,\omega} \iff$ if $Q(w)$ for some $w \in \Sigma^*$ then that there is some $a \in \Sigma$ such that $Q(wa)$.

The next result says that there is a bijective correspondence between prefix-closed and persistent prefix-closed properties also in the case of finite and infinite traces, but that there are exponentially more such properties than in the case of just finite traces:

Proposition 6 $|\text{PersistentPrefixClosed}^{*,\omega}| = |\text{PrefixClosed}^{*,\omega}| = 2^c$.

Proof: We show $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}| \leq |\text{PrefixClosed}^{*,\omega}| \leq 2^c$, where the middle inequality is immediate. For $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}|$, let us define $\varphi : \mathcal{P}((0, 1)) \rightarrow \text{PersistentPrefixClosed}^{*,\omega}$ as

$$\varphi(R) = \bigcup_{0.\alpha \in R} \{\bar{\alpha}\} \cup \text{prefixes}(\bar{\alpha})$$

where we assume for any real number in the interval $(0, 1)$ its decimal binary representation $0.\alpha$ with $\alpha \in \{0, 1\}^\omega$ (if the number is rational then α may contain infinitely many ending 0's), and $\bar{\alpha}$ is the infinite trace in Σ^ω replacing each 0 and 1 in α by $\bar{0}$ and $\bar{1}$, respectively, where $\bar{0}$ and $\bar{1}$ are two arbitrary but fixed distinct elements in Σ (recall that $|\Sigma| \geq 2$). Note that $\varphi(R)$ is well-defined: it is clearly prefix-closed and it is also persistent because its finite traces are exactly prefixes of infinite traces, so they admit continuations in $\varphi(R)$. It is easy to see that φ is injective. Since $|(0, 1)| = c$, we conclude that $2^c \leq |\text{PersistentPrefixClosed}^{*,\omega}|$.

To show $|\text{PrefixClosed}^{*,\omega}| \leq 2^c$, note that any property in $\text{PrefixClosed}^{*,\omega}$ is a union of a subset in Σ^* and a subset in Σ^ω , so $|\text{PrefixClosed}^{*,\omega}| \leq 2^{|\Sigma^*|} \cdot 2^{|\Sigma^\omega|}$. Since $|\Sigma^*| = \aleph_0$, $|\Sigma^\omega| = c$, $2^{\aleph_0} = c$, and $c \cdot 2^c = 2^c$ (by absorption of transfinite cardinals), we get that $|\text{PrefixClosed}^{*,\omega}| \leq 2^c$. \square

The fact that properties in $\text{PersistentPrefixClosed}^{*,\omega}$ contain also infinite traces was crucial in showing the injectivity of φ in the proof above. A similar construction for the finite trace setting does *not* work. Indeed, if

one tries to define a function $\varphi : \mathcal{P}((0, 1)) \rightarrow \text{PersistentSafety}^*$ as $\varphi(R) = \bigcup_{0.\alpha \in R} \text{prefixes}(\bar{\alpha})$, then one can show it well-defined but cannot show it injective: e.g., $\varphi((0, 0.5)) = \varphi((0, 0.5])$.

Since safety properties over finite and infinite traces are governed by the same intuitions as safety properties over only finite or over only infinite traces, the result above tells us that prefix closeness is not a sufficient condition to properly capture the safety properties. Schneider [235] proposes an additional condition in the context of his EM (execution monitoring) framework, namely that if an infinite trace is not in the property, then there is a finite prefix of it which is not in the property either. It is easy to see that this additional condition is equivalent to saying that an infinite trace is in the property whenever all its finite prefixes are in the property, which allows us to compactly define safety properties over finite and infinite traces in the EM style as follows:

Definition 10 $\text{Safety}_{\text{EM}}^{*,\omega} = \{Q \subseteq \Sigma^* \cup \Sigma^\omega \mid u \in Q \text{ iff } \text{prefixes}(u) \subseteq Q\}$.

Note that $\text{Safety}_{\text{EM}}^{*,\omega} \subset \text{PrefixClosed}^{*,\omega}$. We will shortly show that $\text{Safety}_{\text{EM}}^{*,\omega}$ is in fact much smaller than $\text{PrefixClosed}^{*,\omega}$, by showing that $|\text{Safety}_{\text{EM}}^{*,\omega}| = c$.

The consecrated definition of a safety property in the context of both finite and infinite traces is perhaps the one proposed by Lamport in [181], which relaxes the one in Definition 6 by allowing u to range over both finite and infinite traces:

Definition 11 *Let $\text{Safety}^{*,\omega}$ be the set of finite- and infinite-trace properties $\{Q \subseteq \Sigma^* \cup \Sigma^\omega \mid u \notin Q \Rightarrow (\exists w \in \text{prefixes}(u)) (\forall v \in \Sigma^* \cup \Sigma^\omega) wv \notin Q\}$*

Schneider informally stated in [235] that the two definitions of safety above are equivalent. It is not hard to show it formally:

Proposition 7 $\text{Safety}_{\text{EM}}^{*,\omega} = \text{Safety}^{*,\omega}$.

Proof: First note that $\text{Safety}^{*,\omega} \subseteq \text{PrefixClosed}^{*,\omega}$: if $wu \in Q \in \text{Safety}^{*,\omega}$ and $w \notin Q$ then there is some $w' \in \text{prefixes}(w)$, say $w = w'w''$, such that $w'v \notin Q$ for any v , in particular $w'w''u \notin Q$, which contradicts $wu \in Q$.

$\text{Safety}^{*,\omega} \subseteq \text{Safety}_{\text{EM}}^{*,\omega}$: let $Q \in \text{Safety}^{*,\omega}$ and $u \in \Sigma^* \cup \Sigma^\omega$ s.t. $\text{prefixes}(u) \subseteq Q$; if $u \notin Q$ then there is some $w \in \text{prefixes}(u)$ s.t. $wv \notin Q$ for any v , in particular for v the empty word, that is, $w \notin Q$, which contradicts $\text{prefixes}(u) \subseteq Q$.

$\text{Safety}_{\text{EM}}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$: let $u \notin Q \in \text{Safety}_{\text{EM}}^{*,\omega}$; then $\text{prefixes}(u) \not\subseteq Q$, that is, there is some $w \in \text{prefixes}(u)$ s.t. $w \notin Q$; since Q is prefix-closed, it follows that $wv \notin Q$ for any $v \in \Sigma^* \cup \Sigma^\omega$. \square

We next show that there is a bijective correspondence between the safety properties over finite or infinite traces above and the finite trace safety properties in Section 3.1:

Theorem 4 $|\text{Safety}^{*,\omega}| = |\text{Safety}_{\text{EM}}^{*,\omega}| = |\text{Safety}^*| = c$.

Proof: $\text{Safety}^* \subset \text{Safety}_{\text{EM}}^{*,\omega}$ since the properties in $\text{Safety}_{\text{EM}}^{*,\omega}$ are prefix-closed, so $|\text{Safety}^*| \leq |\text{Safety}_{\text{EM}}^{*,\omega}|$.

Since the functions $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ and $\text{prefixes} : \mathcal{P}(\Sigma^\omega) \rightarrow \mathcal{P}(\Sigma^*)$ have actual co-domains Safety^* and $\text{PersistentSafety}^*$, respectively, they can be organized as a function $\text{prefixes} : \text{Safety}_{\text{EM}}^{*,\omega} \rightarrow \text{Safety}^*$. Let us show that this function is injective. Let us assume $Q \neq Q' \in \text{Safety}_{\text{EM}}^{*,\omega}$, say $u \in Q$ and $u \notin Q'$, s.t. $\text{prefixes}(Q) = \text{prefixes}(Q')$. Since $u \in Q \in \text{Safety}_{\text{EM}}^{*,\omega}$ it follows that $\text{prefixes}(u) \subseteq \text{prefixes}(Q) \subseteq Q$, which implies that $\text{prefixes}(u) \subseteq \text{prefixes}(Q') \subseteq Q'$; since $Q' \in \text{Safety}_{\text{EM}}^{*,\omega}$, it follows that $u \in Q'$, contradiction. Therefore, $\text{prefixes} : \text{Safety}_{\text{EM}}^{*,\omega} \rightarrow \text{Safety}^*$ is injective, which proves that $\text{Safety}_{\text{EM}}^{*,\omega} \leq \text{Safety}^*$.

The rest follows by Proposition 7 and Theorem 2. \square

3.4 “Always Past” Characterization

Another common way to specify safety properties is by giving an arbitrary property on finite traces, not necessarily prefix closed, and then to require that any acceptable behavior must have all its finite prefixes in the given property. A particularly frequent case is when one specifies the property of the finite-prefixes using the past-time fragment of linear temporal logics (LTL). For example, Manna and Pnueli [187] call the resulting “always (past LTL)” properties *safety formulae*; many other authors, including ourselves, adopted the terminology “safety formula” from Manna and Pnueli, although some qualify it as “LTL safety formula”. An example of an LTL safety formula is “always (b implies eventually in the past a)”, written using LTL notation as “ $\Box(b \rightarrow \Diamond a)$ ”; here the past time formula “ $b \rightarrow \Diamond a$ ” compactly specifies all the finite-traces

$$\{ws w' s' \mid w, w' \in \Sigma^*, s, s' \in \Sigma, a(s) \text{ and } b(s') \text{ hold}\} \cup \{ws \mid w \in \Sigma^*, s \in \Sigma, b(s) \text{ does not hold}\}.$$

From safety: *We will investigate the case when safety properties are expressed as LTL safety formulae, as well as optimal monitoring techniques for such safety properties, in Section 10.7.*

In the remainder of this section we assume that the past time prefix properties are given as ordinary sets of finite-traces (so we make abstraction of how these properties are expressed) and show not only that the resulting “always past” properties are safety properties, but also that any safety properties can be expressed as an “always past” property. This holds for all the variants of safety properties (i.e., over finite traces, over infinite traces, or over both finite and infinite traces).

Definition 12 *Let $P \subseteq \Sigma^*$ be any property over finite traces. Then we define the “always past” property $\Box P$ as follows:*

- (finite traces) $\{w \in \Sigma^* \mid \text{prefixes}(w) \subseteq P\}$; and*
- (infinite traces) $\{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$; and*
- (finite and infinite traces) $\{u \in \Sigma^* \cup \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\}$.*

Let Safety_\Box^ , $\text{Safety}_\Box^\omega$ and $\text{Safety}_\Box^{*,\omega}$ be the corresponding sets of properties.*

From safety: *In Section 10.7 we show that the language $\mathcal{L}(\Box\varphi)$, that corresponds to the LTL safety formula $\Box\varphi$ for φ some past-time LTL formula, is a property in $\text{Safety}_\Box^\omega$. If one was interested in a finite-trace or in a both finite and infinite trace semantics of LTL, then one could have shown that $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^*$ or that $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^{*,\omega}$.*

Intuitively, one can regard the square “ \Box ” as a closure operator. Technically, it is not precisely a closure operator because it does not operate on the same set: it takes finite-trace properties to any of the three types of properties considered. Since prefixes takes properties back to finite-trace properties, we can show the following result saying that the square is a “closure operator via prefixes ”, and that safety properties are precisely the sets of words which are closed this way:

Proposition 8 *The following hold for all three types of safety properties:*

- $\Box(\text{prefixes}(\Box P)) = \Box P$ for any $P \subseteq \Sigma^*$;
- Q is a safety property iff $\Box(\text{prefixes}(Q)) = Q$.

Proof: Left as an exercise to the reader. See Exercise 5. \square

We next show that the “always past” properties are all safety properties and, moreover, that any safety property can be expressed as an “always past” property:

Theorem 5 *The following hold:*

- $\text{Safety}_{\square}^* = \text{Safety}^*$,
- $\text{Safety}_{\square}^{\omega} = \text{Safety}^{\omega}$, and
- $\text{Safety}_{\square}^{*,\omega} = \text{Safety}^{*,\omega}$.

Therefore, each of the “always past” safety properties have the cardinal c .

Proof: We prove each of the equalities by double inclusion.

$\text{Safety}_{\square}^* \subseteq \text{Safety}^*$. It is true because any property $\square P$ in $\text{Safety}_{\square}^*$ is prefix-closed.

$\text{Safety}^* \subseteq \text{Safety}_{\square}^*$. If $P \in \text{Safety}^*$ then we claim that $P = \square P$, so $P \in \text{Safety}_{\square}^*$. Indeed, since P is prefix-closed, $\text{prefixes}(w) \subseteq P$ for any $w \in P$, so $w \in \square P$; also, since $w \in \text{prefixes}(w)$, it follows that for any $w \in \square P$, $w \in P$.

$\text{Safety}_{\square}^{\omega} \subseteq \text{Safety}^{\omega}$. Let $\square P$ be an “always past” property in $\text{Safety}_{\square}^{\omega}$, and let u be an infinite trace in Σ^{ω} such that $u \notin \square P$. Then it follows that $\text{prefixes}(u) \not\subseteq P$, that is, there is some $w \in \text{prefixes}(u)$ such that $w \notin P$. Since $w \in \text{prefixes}(wv)$ for any $v \in \Sigma^{\omega}$, it means that there is no $v \in \square P$ such that $\text{prefixes}(wv) \subseteq P$, that is, there is no $v \in \Sigma^{\omega}$ such that $wv \in \square P$. Therefore, $\square P \in \text{Safety}^{\omega}$.

$\text{Safety}^{\omega} \subseteq \text{Safety}_{\square}^{\omega}$. If $Q \in \text{Safety}^{\omega}$ then we claim that $Q = \square \text{prefixes}(Q)$. The inclusion $Q \subseteq \square \text{prefixes}(Q)$ is clear, because $u \in Q$ implies $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$. For the other inclusion, note that if $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$ for some $u \in \Sigma^{\omega}$, then u must be in Q : if $u \notin Q$ then by the definition of $Q \in \text{Safety}^{\omega}$, there is some $w \in \text{prefixes}(u)$ which cannot be completed into an infinite trace in Q , which contradicts $\text{prefixes}(u) \subseteq \text{prefixes}(Q)$.

$\text{Safety}_{\square}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$. By Proposition 7, it suffices to show that $\text{Safety}_{\square}^{*,\omega} \subseteq \text{Safety}_{\text{EM}}^{*,\omega}$. Let $\square P$ be an “always past” property in $\text{Safety}_{\square}^{*,\omega}$, and let $u \in \Sigma^* \cup \Sigma^{\omega}$ such that $\text{prefixes}(u) \subseteq \text{prefixes}(\square P)$. Since $\text{prefixes}(\square P) \subseteq P$, it follows that $u \in \square P$; therefore, $\square P \in \text{Safety}_{\text{EM}}^{*,\omega}$.

$\text{Safety}^{*,\omega} \subseteq \text{Safety}_{\square}^{*,\omega}$. It is straightforward to see that $Q \in \text{Safety}_{\text{EM}}^{*,\omega}$ implies $Q = \square \text{prefixes}(Q)$.

The cardinality part follows by Theorems 2, 3, and 4. \square

Proposition 8 and Theorem 5 give yet another characterization for safety properties over any of the three combinations of traces, namely one in the style of the equivalent formulation of safety over infinite traces in Proposition 4: Q is a safety property iff it contains precisely the words whose prefixes are in $\text{prefixes}(Q)$.

Exercises

Exercise 2 The $\text{prefixes} : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$ in Definition 1 is a closure operator: it is extensive ($P \subseteq \text{prefixes}(P)$), monotone ($P \subseteq P'$ implies $\text{prefixes}(P) \subseteq \text{prefixes}(P')$), and idempotent ($\text{prefixes}(\text{prefixes}(P)) = \text{prefixes}(P)$).

Exercise 3 (Counter-)Example 4 showed that the finiteness of Σ was necessary in order for Proposition 2 to hold, by defining a property P over $\Sigma = \mathbb{N} \cup \{\infty\}$ in which all non-empty words start with ∞ . Can we remove ∞ from Σ and from all the words in P ? Why, or why not?

Exercise 4 Same like Exercise 3, but for Example 5 instead of Example 4.

Exercise 5 Prove Proposition 8.

Exercise 6 The “closure under limits” operation in Definition 7 is indeed a closure operator on Σ^ω : it is extensive ($Q \subseteq \overline{Q}$), monotone ($Q \subseteq Q'$ implies $\overline{Q} \subseteq \overline{Q'}$), and idempotent ($\overline{\overline{Q}} = \overline{Q}$).

Chapter 4

Monitoring

In this section we give yet another characterization of safety properties, namely as monitorable properties. Specifically, we formally define a monitor as a (possibly infinite) state machine without final states but with a partial transition function, and then we show that safety properties are precisely the properties that can be monitored with such monitors. We then elaborate on the problem of defining the complexity of monitoring a safety property, discussing some pitfalls and guiding principles, and show that monitoring a safety property can be an arbitrarily hard problem. Finally, we give a more compact and mathematical equivalent definition of a monitor, which may be useful in further foundational efforts in this area.

Relate our definition of a monitor with Schneider's security automata

4.1 Specifying Safety Properties as Monitors

Safety properties are difficult to work with as flat sets of finite or infinite words, not only because they can contain infinitely many words, but also because such a flat representation is inconvenient for further analysis. It is important therefore to *specify* safety properties using formalisms that are easier to represent and reason about.

From safety: *The next sections in this paper investigate several dedicated formalisms that proved to be convenient in specifying safety, such as finite state machines, regular expressions and temporal logics, together with corresponding limitations and efficient monitor synthesis techniques.*

Formalisms known to be useful for specifying safety properties include regular expressions and temporal logics, which can be efficiently translated into finite-state machines which can then be used as monitors. In this section we formalize the intuitive notion of a *monitor* as a special state machine and give yet another characterization of safety properties, namely as *monitorable properties*. Since monitorable properties are completely defined by their monitors, it follows that *all* safety properties can be specified by their corresponding monitors.

Recall that we work under the assumption that Σ is a set of events or program states such that $|\Sigma| \leq \aleph_0$.

Definition 13 *A Σ -monitor, or just a monitor (when Σ is understood), is a triple $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, where S is a set of states, $s_0 \in S$ is the initial state, and M is a deterministic partial transition function.*

Therefore, a monitor as defined above is nothing but a deterministic state machine without final states. Moreover, the set of states is allowed to be infinite, and the transition function has no complexity requirements (it can even be undecidable). We could have defined monitors to be standard state machines, but the subsequent technical developments would have been slightly more involved.

From safety: *In fact, we aim at shortly giving an even more compact definition of a monitor, that we will call canonical monitor, which appears to be sufficient to capture any safety property.*

The intuition for a monitor is the expected one: the monitor is driven by events generated by the observed program (the letters in Σ)—each newly received event drives the monitor from its current state to some other state, as indicated by the transition function M ; if the monitor ever gets stuck, that is, if the transition function M is undefined on the current state and the current event, then the monitored property is declared violated at that point by the monitor.

For any partial function $M : S \times \Sigma \rightarrow S$, we obey the following common notational convention. If $s \in S$ and $w = w_1 w_2 \dots w_k \in \Sigma^*$, we write “ $M(s, w) \downarrow$ ” whenever $M(s, w)$ is defined, that is, whenever $M(s, w_1)$ and $M(M(s, w_1), w_2)$ and ... and $M(\dots(M(s, w_1), w_2) \dots, w_k)$ are all defined, which is nothing but only saying that $M(\dots(M(s, w_1), w_2) \dots, w_k)$ is defined. If we write $M(s, w) = s'$ for some $s' \in S$, then, as expected, we mean that $M(\dots(M(s, w_1), w_2) \dots, w_k)$ is defined and equal to s' .

A monitor specifies a finite-trace property, an infinite-trace property, as well as a finite- and infinite-trace property:

Definition 14 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we define the following properties:*

- $\mathcal{L}^*(\mathcal{M}) = \{w \in \Sigma^* \mid M(s_0, w) \downarrow\}$,
- $\mathcal{L}^\omega(\mathcal{M}) = \{u \in \Sigma^\omega \mid M(s_0, w) \downarrow \text{ for all } w \in \text{prefixes}(u)\}$, and
- $\mathcal{L}^{*,\omega}(\mathcal{M}) = \mathcal{L}^*(\mathcal{M}) \cup \mathcal{L}^\omega(\mathcal{M})$.

We call $\mathcal{L}^*(\mathcal{M})$ the finite-trace property specified by \mathcal{M} , call $\mathcal{L}^\omega(\mathcal{M})$ the infinite-trace property specified by \mathcal{M} , and call $\mathcal{L}^{*,\omega}(\mathcal{M})$ the finite- and infinite-trace property specified by \mathcal{M} . Also, we let

$$\mathcal{S}_{\mathcal{M}} = \{s \in S \mid (\exists w \in \Sigma^*) M(s_0, w) = s\}$$

denote the set of reachable states of \mathcal{M} .

A *monitorable* property is a property which can be specified by a monitor. We next capture this intuitive notion formally:

Definition 15 *For a property $P \subseteq \Sigma^* \cup \Sigma^\omega$, we let $\text{Monitors}(P)$ be the set of monitors $\{\mathcal{M} \mid \mathcal{L}^{*,\omega}(\mathcal{M}) = P\}$. If $\text{Monitors}(P) \neq \emptyset$ then P is called monitorable and the elements of $\text{Monitors}(P)$ are called monitors of P . We define the following classes of properties:*

- $\text{Monitorable}^* = \{P \subseteq \Sigma^* \mid P \text{ monitorable}\}$,
- $\text{Monitorable}^\omega = \{P \subseteq \Sigma^\omega \mid P \text{ monitorable}\}$, and
- $\text{Monitorable}^{*,\omega} = \{P \subseteq \Sigma^* \cup \Sigma^\omega \mid P \text{ monitorable}\}$.

The notion of persistence can also be adapted to monitors:

Definition 16 A monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is persistent iff for any reachable state $s \in \mathcal{S}_{\mathcal{M}}$, there is an $a \in \Sigma$ such that $M(s, a) \downarrow$. Let

- $\text{PersistentMonitorable}^* = \{\mathcal{L}^*(\mathcal{M}) \mid \mathcal{M} \text{ persistent}\}$

be the set of finite-trace properties monitorable by persistent monitors.

Our next goal is to show that each monitor admits a largest persistent “submonitor”. To formalize it, we lift the conventional partial order relation on partial functions to monitors:

Definition 17 If $\mathcal{M}_1 = (S, s_0, M_1 : S \times \Sigma \rightarrow S)$ and $\mathcal{M}_2 = (S, s_0, M_2 : S \times \Sigma \rightarrow S)$ are two monitors sharing the same states and initial state, then let $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$, read \mathcal{M}_1 a submonitor of \mathcal{M}_2 , iff for any $s \in S$ and any $a \in \Sigma$, if $M_1(s, a)$ is defined then $M_2(s, a)$ is also defined and $M_2(s, a) = M_1(s, a)$.

The above can be easily generalized to allow \mathcal{M}_1 to only have a subset of the states of \mathcal{M}_2 , but we found that generalization unnecessary so far.

The above partial-order on monitors allows us to use conventional mathematics to obtain the largest persistent sub-monitor of a monitor:

Proposition 9 $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M} \text{ and } \mathcal{K} \text{ persistent}\}, \sqsubseteq)$ is a complete (join) semilattice for any monitor \mathcal{M} .

Proof: If $\{\mathcal{K}_i = (S, s_0, K_i : S \times \Sigma \rightarrow S) \in \mathcal{M}\}_{i \in I}$ is a set of persistent monitors, then their supremum (or join) is the monitor $\mathcal{K} = (S, s_0, K : S \times \Sigma \rightarrow S)$ where $K(s, a) = s'$ iff there is some $i \in I$ such that $K_i(s, a) = s'$. It is easy to see that \mathcal{K} is a well-defined monitor and that it is persistent. \square

Since complete semilattices have maximum elements, the following definition is fully justified:

Definition 18 For any monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we let $\mathcal{M}^\circ = (S, s_0, M^\circ : S \times \Sigma \rightarrow S)$ be the \sqsubseteq -maximal element of the complete lattice $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M} \text{ and } \mathcal{K} \text{ persistent}\}, \sqsubseteq)$.

We next show that, as expected, there is a tight relationship between persistent safety properties (Definition 3) and persistent canonical monitors.

Proposition 10 Let $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$. Then the following hold:

- $\mathcal{L}^\omega(\mathcal{M}) = \mathcal{L}^\omega(\mathcal{M}^\circ)$,

- $\mathcal{L}^*(\mathcal{M}^\circ) = \mathcal{L}^*(\mathcal{M})^\circ$, and
- \mathcal{M} persistent iff $\mathcal{L}^*(\mathcal{M})$ persistent.

Proof: The first property can be shown by the following sequence of equivalences: $u \in \mathcal{L}^\omega(\mathcal{M})$ iff $M(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff there is some persistent monitor $\mathcal{K} \sqsubseteq \mathcal{M}$ such as $K(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff $M^\circ(s_0, w) \downarrow$ for all $w \in \text{prefixes}(u)$, iff $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$.

The second property can be shown as follows: $w \in \mathcal{L}^*(\mathcal{M}^\circ)$ iff $M^\circ(s_0, w) \downarrow$, iff there is some $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$ such that $w \in \text{prefixes}(u)$ (because \mathcal{M}° is persistent), iff there is some $u \in \mathcal{L}^\omega(\mathcal{M})$ such that $w \in \text{prefixes}(u)$ (by the first property), iff there is some $u \in \Sigma^\omega$ such that $w \in \text{prefixes}(u) \subseteq \mathcal{L}^*(\mathcal{M})$, iff there is some $u \in \Sigma^\omega$ such that $w \in \text{prefixes}(u) \subseteq \mathcal{L}^*(\mathcal{M})^\circ$ (because $\text{prefixes}(u)$ is a persistent safety property), iff $w \in \mathcal{L}^*(\mathcal{M})^\circ$.

Finally, the third property is an immediate consequence of the second, noticing that \mathcal{M} is persistent iff it is equal to \mathcal{M}° , and that $\mathcal{L}^*(\mathcal{M})$ is persistent iff it is equal to $\mathcal{L}^*(\mathcal{M})^\circ$. \square

Theorem 6 *The following hold:*

- $\text{Monitorable}^* = \text{Safety}^*$,
- $\text{Monitorable}^\omega = \text{Safety}^\omega$,
- $\text{Monitorable}^{*,\omega} = \text{Safety}^{*,\omega}$, and
- $\text{PersistentMonitorable}^* = \text{PersistentSafety}^*$.

Proof: First, note that the following hold for any monitor \mathcal{M} :

- $\mathcal{L}^*(\mathcal{M}) \in \text{Safety}^*$,
- $\mathcal{L}^\omega(\mathcal{M}) \in \text{Safety}^\omega$, and
- $\mathcal{L}^{*,\omega}(\mathcal{M}) \in \text{Safety}^{*,\omega}$.

These all follow by Theorem 5: taking P in Definition 12 to be the property $\{w \in \Sigma^* \mid M(s_0, w) \downarrow\}$, then $\Box P$ over finite traces is precisely $\mathcal{L}^*(\mathcal{M})$, over infinite traces is precisely $\mathcal{L}^\omega(\mathcal{M})$, and over finite and infinite traces is precisely $\mathcal{L}^{*,\omega}(\mathcal{M})$, so the three languages are in Safety_\square^* , $\text{Safety}_\square^\omega$, and $\text{Safety}_\square^{*,\omega}$, respectively. Therefore, $\text{Monitorable}^* \subseteq \text{Safety}^*$, $\text{Monitorable}^\omega \subseteq \text{Safety}^\omega$, and $\text{Monitorable}^{*,\omega} \subseteq \text{Safety}^{*,\omega}$.

Second, note that we can associate a default monitor \mathcal{M}_P to any finite-trace property $P \subseteq \Sigma^*$, namely $(S_P, \epsilon, M_P : S_P \times \Sigma \rightarrow S_P)$, where $S_P = \text{prefixes}(P)$, ϵ is the empty word, and $M_P(w, a)$ is defined iff $wa \in \text{prefixes}(P)$, and in that case $M_P(w, a) = wa$. Moreover, it is easy to check that

- $\mathcal{L}^*(\mathcal{M}_P) = \{w \in \Sigma^* \mid \text{prefixes}(w) \subseteq P\} = \Box P$ (over finite traces) ,
- $\mathcal{L}^\omega(\mathcal{M}_P) = \{u \in \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\} = \Box P$ (over infinite traces),
- $\mathcal{L}^{*,\omega}(\mathcal{M}_P) = \{u \in \Sigma^* \cup \Sigma^\omega \mid \text{prefixes}(u) \subseteq P\} = \Box P$ (over both finite and infinite traces).

Since P was chosen arbitrarily, it follows then by Theorem 5 that $\text{Safety}^* \subseteq \text{Monitorable}^*$, $\text{Safety}^\omega \subseteq \text{Monitorable}^\omega$, and $\text{Safety}^{*,\omega} \subseteq \text{Monitorable}^{*,\omega}$.

Finally, the equality $\text{PersistentMonitorable}^* = \text{PersistentSafety}^*$ follows by the first fact and by Proposition 10. \square

4.2 Complexity of Monitoring a Safety Property

We here address the problem of defining the complexity of monitoring. Before we give our definition, let us first discuss some pitfalls in defining this notion. Our definition for the complexity of monitoring resulted as a consequence of trying to avoid these pitfalls. Let P be a safety property.

Pitfall 1.

The complexity of monitoring P is nothing but the complexity of checking, for an input word $w \in \Sigma^$, whether $w \in \text{prefixes}(P)$.*

This would be an easy to formulate decision problem, but, unfortunately, does not capture well the intuition of monitoring, because it does not require that the word w be processed incrementally, as its letters become available from the observed system. Incremental processing of letters can make a huge difference in both how complex monitoring is and how monitoring complexity can be defined. For example, it is well-known that the membership problem of a finite word to the language of an extended regular expression (ERE), i.e., a regular expression extended with complement operators, is a polynomial problem (the classic algorithm by Hopcroft and Ullman [148] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$, where m is the size of the word and n that of the expression). However,

From safety: as shown in Section 7

there are EREs defining safety properties whose monitoring requires non-elementary space and time. Of course, this non-elementary lower-bound is expressed only as a function of the size of the ERE representing the safety property; it does not take into account the size of the monitored trace. This leads us to our first guiding principle:

Principle 1.

The complexity of monitoring a safety property P should depend only upon P , not upon the trace being monitored.

Indeed, since monitoring is a process that involves potentially unbounded traces, if the complexity of monitoring a property P were expressed as a function of the execution trace as well, then that complexity measure would be close to meaningless in practice, because monitoring reactive systems would have unbounded complexity. For example, consider an operating system monitoring some safety property on how its resources are being used by the various running processes; what one would like to know here is what is the runtime overhead of monitoring that safety property at each relevant event, and not the obvious fact that the more the operating system runs the larger the total runtime overhead is.

Nevertheless, one can admittedly argue that it would still be useful to know how complex the monitoring of P against a given finite trace w is, in terms of both the size of (some representation of) P and the size of w ; however, this is nothing but a conventional membership test decision problem, that has nothing to do with monitoring. If one picks some arbitrary off-the-shelf efficient algorithm for membership testing and uses that at each newly received event on the existing finite execution trace, then one may obtain a “monitoring” algorithm whose complexity to process each event grows in time, as events are processed. In the context of monitoring a reactive system, that means that eventually the monitoring process may become unfeasible, regardless of how many resources are initially available and regardless of how efficient the membership testing algorithm is. What one needs in order for the monitoring process to stay feasible regardless of how many events are observed, is a special membership algorithm that processes each event as received and whose state or processing time does not increase potentially unbounded as events are received. Therefore, one needs an algorithm which, if it takes resources R to check w , then it takes at most $R + \Delta$ to check a one-event continuation wa of w , where Δ *does not depend on w* . In other words, one needs a *monitor for P of complexity Δ* .

Pitfall 2.

P is typically infinite, so the complexity of monitoring P should be a function of the size of some finite specification, or representation, of P.

Indeed, since Principle 1 tells us that the complexity of monitoring P is a function of P only and not of the monitored trace, one may be tempted to conclude that it is a function of the *size* of some convenient encoding of P . There are at least two problems with this approach, that we discuss below.

- One problem is that the same property P can be specified in many different ways as a structure of finite size; for example, it can be specified as a regular expression, as an extended regular expression, as a temporal logic formula, as an ordinary automaton, as a push-down automaton, etc. These formalisms may represent P as specifications of quite different sizes. Which is the most appropriate? It is, nevertheless, interesting and important to study the complexity of monitoring safety properties expressed using different specification formalisms, as a function of the property representation size, because that can give us an idea of the amount of resources needed to monitor a particular specification. However, one should be aware that such a complexity measure is an attribute of the corresponding specification formalisms, not of the specified property itself. Indeed, the higher this complexity measure for a particular formalism, the higher the encoding strength of safety properties in that formalism: for example, the complexity of monitoring safety properties expressed as EREs is non-elementary in the size of the original ERE, while the complexity of monitoring the same property expressed as an ordinary regular expression is linear in the size of the regular expression. Does that mean that one can monitor safety properties expressed as regular expressions non-elementarily more efficiently than one can monitor safety properties expressed as EREs? Of course not, because EREs and regular expressions have the same expressiveness, so they specify exactly the same safety properties. All it means is that EREs can express safety-properties non-elementarily more compactly than ordinary regular expressions.
- Another problem with this approach is that apparently appropriate representations of P may be significantly larger than it takes to monitor P . One may say, for example, that, whenever possible, a natural way to specify a particular safety property is as a finite-state machine, e.g.,

as a monitor like in Definition 107 . To be more concrete, consider that the safety property P_n saying “every 2^n -th event is a ” is specified as a monitor of 2^n states that transits with any event from each state to the next one, except for the 2^n -th state, which has only one transition, with event a , back to state 1. Therefore, the size of this representation of P_n is $\Omega(2^n)$. Assuming that each state takes n bits of storage (for example, assume that states are exactly the binary encodings of the numbers 1, 2, 3, ..., 2^n) and that the next state can be calculated from the current state in linear complexity with the size of the state (which is true in our scenario), then it is clear that the actual complexity of monitoring P_n is $O(n)$. If the complexity of monitoring P_n were a function of the size of the specification of P_n , then one could wrongly conclude that the complexity of monitoring “every 2^n -th event is a ” is $O(2^n)$.

Therefore, a safety property P has an inherent complexity w.r.t. monitoring, complexity which has nothing to do with how P is represented, or encoded, or specified. It is that inherent complexity attribute of safety properties that we are after here. From the discussion above, we draw our second guiding principle:

Principle 2.

The monitoring complexity of a safety property P is an attribute of P alone, not a function of the size of some adhoc representation of P .

By Theorem 6, safety properties are precisely those properties that are monitorable, that is, those properties P for which there are (finite-state or not) monitors $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ whose (finite-trace, infinite-trace, or finite- and infinite-trace—this depends upon the type of P) language is precisely P . Any algorithm, program or system that one may come up with to be used as a monitor for P , can be organized as a monitor of the form $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ for P . Consequently, the complexity of monitoring P cannot be smaller than the functional complexity of the partial function $M : S \times \Sigma \rightarrow S$ corresponding to some “best” monitor \mathcal{M} for P ; if there are no additional restrictions, then by “best” monitor we mean the one whose functional complexity of M is smallest. In particular, if there is no monitor for P whose transition partial function M is decidable, then we can safely say that the problem of monitoring P is undecidable. This discussion leads to the following:

Pitfall 3.

The complexity of monitoring P is the functional complexity of function M , where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is the “best” monitor for P .

Since safety properties are precisely the monitorable properties, this appears to be a very natural definition for the complexity of monitoring. While the functional complexity of the monitor function is indeed important because it directly influences the efficiency of monitoring, it is *not* a sufficient measure for the complexity of monitoring. That is because the functional complexity of M only says how complex M is in terms of the size of *its input*; it does not say anything about how large the state of the monitor can grow in time. For example, the rewriting-based monitoring algorithm for EREs from [219], whose states are EREs and whose transition is a derivative operation of functional complexity $O(n^2)$ taking an ERE of size n into an ERE of size $O(n^2)$. It would be very misleading to say that the complexity of monitoring EREs is $O(n^2)$, because it may sound much better than it actually is: the n^2 factor accumulates as events are processed. Any monitor for EREs, including the one based on derivatives, eventually requires non-elementary resources (in the size of the ERE) to process a new event.

Therefore, while the complexity of the function M being executed at each newly received event by a monitor \mathcal{M} is definitely a necessary and important factor to be considered when defining the complexity of monitoring using \mathcal{M} , it is not sufficient. One also needs to take into account the size of the input that is being passed to the monitoring function, that is, the size of the monitor state together with the size of the received event. In particular, a monitor storing all the observed trace has unbounded complexity, say ∞ , even though its monitoring function has trivial complexity (e.g., the “event storing” function has linear complexity). More generally, if a property admits no finite-state monitor, then we’d like to say that its monitoring complexity is ∞ : indeed, for any monitor for such a property and for any amount of resources R , there is some sequence of events that would lead the monitor to a state that needs more than R resources to be stored or computed. These observations lead us to the following:

Principle 3. The complexity of monitoring P is a function of both the functional complexity of M and of the size of the states in S , where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ is an appropriately chosen (“best”) monitor for P .

We next follow the three principles above and derive our definition for the complexity of monitoring a safety property P . Before that, let us first define the complexity of monitoring a safety property using a particular monitor for that property, or in other words, let us first define the complexity of a monitor.

During a monitoring session using a monitor, at any moment in time one needs to store at least one state, namely the state that the monitor is currently in. When receiving a new event, the monitor launches its transition function on the current state and the received input. Therefore, the (worst-case) complexity of monitoring with $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$ could be defined as

$$\max\{FC(M(s, a)) \mid s \in S, a \in \Sigma\},$$

where $FC(M(s, a))$ is the functional complexity of evaluating M on state s and event a , as a function of the sizes of s and a . In other words, the worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair (state,event). In order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events. Since in order to distinguish N elements one needs $\log(N)$ space, we deduce that one needs at least $\log(|S|)$ space to store the state of the monitor in its worst-case monitoring scenario (each state in S is reachable).

Definition 19 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$, we define the complexity of monitoring \mathcal{M} , written $C_{Mon}(\mathcal{M})$, as the function*

$$FC(M)(\log |S|) : \mathbb{N} \rightarrow \mathbb{N},$$

which is the “uncurried” version applied on $\log |S|$ of the worst-case functional complexity $FC(M) : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ of the partial function M as a function of the size of the pair (state,event) being passed to it.

We assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, “best” monitor for P :

$$\min\{\max\{FC(M(s, a)) \mid s \in S, a \in \Sigma\} \mid \mathcal{M} = (S, s_0, M) \in \text{Monitors}(P)\},$$

From safety: where $FC(M(s, a))$ is the functional complexity of evaluating M on state s and event a , as a function of the sizes of s and a . In other words, we assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, “best” monitor for P . The worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair $(state, event)$. Therefore, in order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events.

This gives us the following:

Definition 20 We let

$$C_{Mon}(P) = \min\{FC(M) \circ \langle \log(|S|), 1_\Sigma \rangle \mid \mathcal{M} = (S, s_0, M) \in \text{Monitors}(P)\}$$

be the complexity of monitoring a safety property P .

4.3 Monitoring Safety Properties is Arbitrarily Hard

We show that the problem of monitoring a safety property can be arbitrarily complex. The previous section tells us that there are as many safety properties as real numbers. Therefore, it is not surprising that some of them can be very hard or impossible to monitor. In this section we formalize this intuitive argument. Our approach is to show that we can associate a safety property P_S to any set of natural numbers S , such that monitoring that safety property is as hard as checking membership of arbitrary natural numbers to S . The result then follows from the fact that checking memberships of natural numbers to sets of natural numbers is a problem that can be arbitrarily complex.

Theorem 2 indirectly says that we can associate a persistent safety property to any set of natural numbers (sets of natural numbers are in a bijective correspondence with the real numbers). However, it is not clear how that safety property looks and neither how to monitor it. We next give a more concrete mapping from sets of natural numbers to (persistent) safety properties and show that monitoring the property is equivalent to

testing membership to the set. It suffices to assume that Σ contains only two elements, say $\Sigma = \{0, 1\}$.

Definition 21 Let $P_- : \mathcal{P}(\mathbb{N}) \rightarrow \text{PersistentSafety}^*$ be the mapping defined as follows: for any $S \subseteq \mathbb{N}$, let P_S be the set $1^* \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^*$.

It is easy to see that P_S is a persistent safety property over finite traces. Also, it is easy to see that the bijection in the proof of Theorem 3 associates to P_S the safety property over infinite traces $1^\omega \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^\omega$.

Let us now investigate the problem of monitoring P_S .

Proposition 11 For any $S \subseteq \mathbb{N}$, monitoring P_S is equivalent to deciding membership of natural numbers to S .

Proof: If M_S is an oracle deciding membership of natural numbers to S , that is, if $M_S(n)$ is true iff $n \in S$, then one can build a monitor for P_S as follows: for a given trace, incrementally read and count the number of prefix 1's; if no 0 is ever observed then monitor indefinitely without reporting any violation; when a first 0 is observed, if any, ask if $M(k)$, where k is the number of 1's observed; if $M(k)$ is false, then report violation; if $M(k)$ is true, then continue monitoring indefinitely and never report violation. It is clear that this is indeed a monitor for P_S .

Conversely, if we had any monitor for P_S then we could build a decision procedure for membership to S as follows: given $k \in \mathbb{N}$, send to the monitor a sequence of k ones followed by a 0; if the monitor reports violation then deduce that $k \notin S$; if the monitor does not report violation, then deduce that $k \in S$. It is clear that this is a decision procedure for membership to S .

The proof works for both persistent safety properties over finite traces and for safety properties over infinite traces. \square

The claim in the title of this section follows now from the fact that the set S of natural numbers can be chosen so that its membership problem is arbitrarily complex. For example, since there are as many subsets of natural numbers as real numbers while there are only as many Turing machines as natural numbers, it follows that there are many (exponentially) more sets of natural numbers that are not recognized by Turing machines than those that are. In particular, there are sets of natural numbers corresponding to any degree in the arithmetic hierarchy, i.e., to predicates $A(k)$ of the form $(Q_1 k_1)(Q_2 k_2) \cdots (Q_n k_n) R(k, k_1, k_2, \dots, k_n)$, where Q_1, Q_2, \dots, Q_n are alternating (universal or existential) quantifiers and R is a recursive/decidable

relation: for A such a predicate, let S_A be the set of natural numbers $\{k \mid A(k)\}$. Recall that if Q_1 is \forall then A is called a Π_n property, while if Q_1 is \exists then A is called a Σ_n property. In particular, $\Sigma_0 = \Pi_0$ and they contain precisely the recursive/decidable properties, Σ_1 contains precisely the class of recursively enumerable problems, Π_1 contains precisely the co-recursively enumerable problems, etc.; a standard Π_2 problem is TOTALITY: given $k \in \mathbb{N}$, is it true that Turing machine with Gödel number k terminates on all inputs? Since each level in the arithmetic hierarchy contains problems strictly harder than problems on the previous layer (because $\Sigma_n \cup \Pi_n \subsetneq \Sigma_{n+1} \cap \Pi_{n+1}$), the arithmetic hierarchy gives us a universe of safety properties whose monitoring can be arbitrarily hard.

Within the decidable fragment, as expected, monitoring safety properties can also have any complexity. Indeed, pick for example any NP-complete problem and let S be the set of inputs (coded as natural numbers) for which the problem has a positive answer; then, as explained in the proof of Proposition 11, monitoring P_S against input 1^k0 is equivalent to deciding membership of k to S , which is further equivalent to answering the NP-complete problem on input k . Of course, in practice a particular (implementation of a) monitor can be more complex than the corresponding membership problem; for example, monitors corresponding to NP-complete problems are most likely exponential. Also, note that a monitor for P_S needs not necessarily do its complex computation on an input 1^k0 when it encounters the 0. It can perform intermediate computations as it reads the prefix 1's and thus pay a lesser computational price when the 0 is encountered. What Proposition 11 says is that the *total* complexity to process the input 1^k0 can be no lower than the complexity of checking whether $k \in S$.

4.4 Canonical Monitors

We conclude this section with an alternative definition of a monitor, called *canonical monitor*, which is more compact than our previous definition and which appears to be sufficient to capture any safety property. We do not make any use of this alternative definition in this paper, but it may serve as a basis for further foundational endeavors in this area.

The set of states S of a monitor $(S, s_0, M : S \times \Sigma \rightarrow S)$ are typically enumerable, so they can be very well replaced with natural numbers. Moreover, the initial state s_0 can be encoded, by convention, as the first natural number, 0. A monitor then becomes nothing but a partial function $\mathbb{N} \times \Sigma \rightarrow \mathbb{N}$. We

therefore rightfully call these particular monitors *canonical*:

Definition 22 A canonical Σ -monitor is a partial function $\mathcal{N} : \mathbb{N} \times \Sigma \rightarrow \mathbb{N}$. Let $\mathcal{S}_{\mathcal{N}} = \{n \mid (\exists w) \mathcal{N}(0, w) = n\}$ be the states of \mathcal{N} . As before, let

- $\mathcal{L}^*(\mathcal{N}) = \{w \in \Sigma^* \mid \mathcal{N}(0, w) \downarrow\}$,
- $\mathcal{L}^\omega(\mathcal{N}) = \{u \in \Sigma^\omega \mid \mathcal{N}(0, w) \downarrow \text{ for all } w \in \text{prefixes}(u)\}$, and
- $\mathcal{L}^{*,\omega}(\mathcal{N}) = \mathcal{L}^*(\mathcal{N}) \cup \mathcal{L}^\omega(\mathcal{N})$.

Although the set of states S in a monitor $(S, s_0, M : S \times \Sigma \rightarrow S)$ is allowed to have any cardinal while the states in canonical monitors are restricted to natural numbers, it turns out that canonical monitors can in fact express all monitorable properties:

Proposition 12 A property $P \subseteq \Sigma^*$ (resp. $P \subseteq \Sigma^\omega$, resp. $P \subseteq \Sigma^* \cup \Sigma^\omega$) is monitorable iff there is some canonical monitor \mathcal{N} such that $P = \mathcal{L}^*(\mathcal{N})$ (resp. $P = \mathcal{L}^\omega(\mathcal{N})$, resp. $P = \mathcal{L}^{*,\omega}(\mathcal{N})$).

Proof: Since any canonical monitor is a monitor, it follows that any property specifiable by a canonical monitor is indeed monitorable. For the converse, let P be a property monitorable by some monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightarrow S)$. Since $|\Sigma| \leq \aleph_0$, we can enumerate all the states of \mathcal{M} that can be reached from s_0 with its transition function M . There are many different ways to do this (e.g., in breadth-first order, in depth-first order, etc.), but these are all ultimately irrelevant. If we let $S^r = \{s_0, s_1, s_2, \dots\}$ denote the resulting set of reachable states, then it is easy to first note that the monitor $\mathcal{M}^r = (S^r, s_0, M : S^r \times \Sigma \rightarrow S^r)$ specifies the same property P as \mathcal{M} , and second note that \mathcal{M}^r specifies the same property as the canonical monitor $\mathcal{N} : \mathbb{N} \times \Sigma \rightarrow \mathbb{N}$ defined by $\mathcal{N}(i, a) = j$ iff $M(s_i, a) = s_j$. \square

Exercises

Exercise 7 Define a canonical monitor for the property

“A file can only be accessed if it is open.”

That is, the file can only be accessed if it was opened at some moment in the past and it was not closed since then. Suppose Σ consists of the events/actions $\{o, a, c\}$, where o stands for “file open”, a for “file access”, and c for “file close”.

Chapter 5

Event/Trace Observation

Chapter 6

Monitor Synthesis: Finite State Machines (FSM)

Currently, this chapter has a lot of duplication. The material has been collected from 2 papers, but not rationally merged yet.

discuss also DFA, NFA, using NFA as monitor, REs, RE2NFA, derivatives

6.1 Binary Transition Trees (BTT)

6.1.1 Multi-Transition and Binary Transition Tree Finite State Machines

To keep the runtime overhead of monitors low, it is crucial to do as little computation as possible in order to proceed to the next state. In the sequel we assume that finite state monitors are desired to efficiently change their state (when a new event is received) to one of possible states s_1, s_2, \dots, s_n , under the knowledge that a transition to each such state is enabled deterministically by some Boolean formula, p_1, p_2, \dots, p_n , respectively, on atomic state predicates.

Definition 23 *Let S be a set whose elements are called states, and let A be another set, whose elements are called atomic predicates. Let $\{s_1, s_2, \dots, s_n\} \subseteq S$ and let p_1, p_2, \dots, p_n be propositions over atoms in A (using the usual*

Boolean combinators presented in Subsection 2.1.2), with the property that exactly one of them is true at any moment, that is, $p_1 \vee p_2 \vee \dots \vee p_n$ holds and for any distinct p_i, p_j , it is the case that $p_i \rightarrow \neg p_j$ holds. Then we call the n -tuple $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ a multi-transition (MT) over states S and atomic predicates (or simply atoms) A . Let $MT(S, A)$ be the set of multi-transitions over states S and atoms A .

Since the rest of the paper is concerned with rather theoretical results, from now on we use mathematical symbols for the Boolean operators instead of ASCII symbols like in Subsection 2.1.2. The intuition underlying multi-transitions is straightforward: depending on which of the propositions p_1, p_2, \dots, p_n is true at a given moment, a corresponding transition to exactly one of the states s_1, s_2, \dots, s_n will take place. Formally,

Definition 24 Maps $\theta : A \rightarrow \{\text{true}, \text{false}\}$ are called events from now on. With the notation above, given an event θ , we define a map $\theta_{MT} : MT(S, A) \rightarrow S$ as $\theta_{MT}([p_1?s_1, p_2?s_2, \dots, p_n?s_n]) = s_i$, where $\theta(p_i) = \text{true}$; notice that $\theta(p_j) = \text{false}$ for any $1 \leq j \neq i \leq n$.

Definition 25 Given an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$, let e_θ denote the list of atomic predicates a with $\theta(a) = \text{true}$. There is an obvious correspondence between events as maps $A \rightarrow \{\text{true}, \text{false}\}$ and events as lists of atomic predicates, which justifies our implementation of events in Subsection 2.1.2. We take the liberty to use either the map or the list notation for events from now on in the paper. We let \mathcal{E} denote the set of all events and call lists, or words, $e_{\theta_1} \dots e_{\theta_n} \in \mathcal{E}^*$ (finite) traces; this is also consistent with our mechanical Maude ASCII notation in Subsection 2.1.2, except that we prefer not to separate events by commas in traces from now on, to avoid mathematical notational conflicts.

We next define binary transition trees.

Definition 26 Under the same notations as in the previous definition, a binary transition tree (BTT) over states S and atoms A is a term over syntax

$$BTT ::= S \mid (A ? BTT : BTT).$$

We let $BTT(S, A)$ denote the set of binary transition trees over states S and atoms A . Given an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$, we define a map

$\theta_{BTT} : BTT(S, A) \rightarrow S$ inductively as follows:

$$\begin{aligned}\theta_{BTT}(s) &= s \text{ for any } s \in S, \\ \theta_{BTT}(a ? b_1 : b_2) &= \theta_{BTT}(b_1) \text{ if } \theta(a) = \text{true}, \text{ and} \\ \theta_{BTT}(a ? b_1 : b_2) &= \theta_{BTT}(b_2) \text{ if } \theta(a) = \text{false}.\end{aligned}$$

A binary transition tree b in $BTT(S, A)$ is said to implement a multi-transition t in $MT(S, A)$ if and only if $\theta_{BTT}(b) = \theta_{MT}(t)$ for any map $\theta : A \rightarrow \{\text{true}, \text{false}\}$.

BTTs generalize BDDs [46], which can be obtained by taking $S = \{\text{true}, \text{false}\}$. As an example of BTT, $a1 ? a2 ? s1 : a3 ? \text{false} : s2 : a3 ? s2 : \text{true}$ says “eval a_1 ; if a_1 then (eval a_2 ; if a_2 then go to state $s1$ else (eval a_3 ; if a_3 then go to state false else go to state $s2$)) else (eval a_3 ; if a_3 then go to state $s2$ else go to state true)”. Note that true and false are just some special states. Depending on the application they can have different meanings, but in our applications true typically means that the monitoring requirement has been fulfilled, while false means that it has been violated. It is often convenient to represent BTTs graphically, such as the one in Figure 6.1.

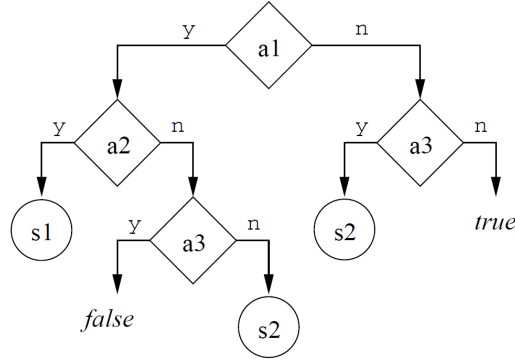


Figure 6.1: Graphical representation for the BTT $a1 ? a2 ? s1 : a3 ? \text{false} : s2 : a3 ? s2 : \text{true}$.

Definition 27 A multi-transition finite state machine (*MT-FSM*) is a triple (S, A, μ) , where S is a set of states, A is a set of atomic predicates, and μ is a map from $S - \{\text{true}, \text{false}\}$ to $MT(S, A)$. If S contains true and/or false , then $\mu(\text{true}) = [\text{true} ? \text{true}]$ and/or $\mu(\text{false}) = [\text{true} ? \text{false}]$, respectively. A binary

transition tree finite state machine (*BTT-FSM*) is a triple (S, A, β) , where S and A are like before and β is a map from $S - \{\text{true}, \text{false}\}$ to $BTT(S, A)$. If S contains *true* and/or *false*, then it is the case that $\beta(\text{true}) = [\text{true}]$ and/or $\beta(\text{false}) = [\text{false}]$, respectively. For an event $\theta : A \rightarrow \{\text{true}, \text{false}\}$ in any of these machines, we let $s \xrightarrow{\theta} s'$ denote the fact that $\theta_{MT}(\mu(s)) = s'$ or $\theta_{BTT}(\beta(s)) = s'$, respectively. We take the liberty to call $s \xrightarrow{\theta} s'$ a transition. Also, we may write $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} s_{n+1}$ and call it a sequence of transitions whenever $s_1 \xrightarrow{\theta_1} s_2, \dots, s_n \xrightarrow{\theta_n} s_{n+1}$ are transitions.

Note that S is not required to contain the special states *true* and *false*, but if it contains them, these states transit only to themselves. The finite state machine notions above are intended to abstract the intuitive concept of a monitor. The states in S are monitor states, and some of them can trigger side effects in practice, such as messages sent or reported to users, actions taken, feedback sent to the monitored program for guidance purposes, etc.

Definition 28 If $\text{true} \in S$ then a sequence of transitions $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} \text{true}$ is called a *validating sequence* (for s_1); if $\text{false} \in S$ then $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n} \text{false}$ is called an *invalidating sequence* (for s_1). These notions naturally extend to corresponding finite traces of events $e_{\theta_1} \dots e_{\theta_n}$.

BTT-FSMs can and should be thought of as efficient “implementations” of MT-FSMs, in the sense that they implement multi-transitions as binary transition trees. These allow one to reduce the amount of computation in a monitor implementing a BTT-FSM to only evaluate at most all the atomic predicates in a given state in order to decide to which state to go next; however, it will only very rarely be the case that *all* the atomic predicates need to be evaluated.

A natural question is why one should bother at all then with defining and investigating MT-FSMs. Indeed, what one really looks for in the context of FSM monitoring is efficient BTT-FSMs. However, MT-FSMs are nevertheless an important intermediate concept as it will become clearer later in the paper. This is because they have the nice property of *state mergeability*, which allows one to elegantly generate MT-FSMs from logical formulae. By state mergeability we mean the following. Suppose that during a monitor generation process, such as that for LTL that will be presented in Subsection 8.4.1, one proves that states s and s' are “logically equivalent” (for now, equivalence can be interpreted intuitively: they have the same behavior),

and that s and s' have the multi-transitions $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ and $[p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]$. Then we can merge s and s' into one state whose multi-transition is $Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}])$, defined as follows:

$$\begin{aligned}
& Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]) \\
& \text{contains all choices } p?s'', \text{ where } s'' \text{ is a state in } \{s_1, s_2, \dots, s_n\} \cup \\
& \{s'_1, s'_2, \dots, s'_{n'}\} \text{ and} \\
& \begin{aligned}
& \bullet p \text{ is } p_i \text{ when } s'' = s_i \text{ for some } 1 \leq i \leq n \text{ and } s'' \neq s'_{i'} \text{ for all} \\
& \quad 1 \leq i' \leq n', \text{ or} \\
& \bullet p \text{ is } p'_{i'} \text{ when } s'' = s'_{i'} \text{ for some } 1 \leq i' \leq n' \text{ and } s'' \neq s_i \text{ for} \\
& \quad \text{all } 1 \leq i \leq n, \text{ or} \\
& \bullet p \text{ is } p_i \vee p'_{i'} \text{ when } s'' = s_i \text{ for some } 1 \leq i \leq n \text{ and } s'' = s'_{i'} \\
& \quad \text{for some } 1 \leq i' \leq n'.
\end{aligned}
\end{aligned}$$

It is easy to see that this multi-transition merging operation is well defined, that is,

Proposition 13 *$Merge([p_1?s_1, p_2?s_2, \dots, p_n?s_n], [p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}])$ is a well-formed multi-transition whenever both $[p_1?s_1, p_2?s_2, \dots, p_n?s_n]$ and $[p'_1?s'_1, p'_2?s'_2, \dots, p'_n?s'_{n'}]$ are well-formed multi-transitions. Therefore, MERGE can be seen as a function $\mathcal{P}_f(MT(S, A)) \rightarrow MT(S, A)$, where \mathcal{P}_f is the finite powerset operator.*

There are situations when a definite answer, *true* or *false* is desired at the end of the monitoring session, as it will be in the case of LTL. As explained in Section 8.1, the intuition for the last event in an execution trace is that the trace is infinite and stationary in that last event. This seems to be the best and simplest assumption about future when a monitoring session is ended.

Discussion about variants of finite-trace LTL needed.

For such situations, we enrich our definitions of MT-FSM and BTT-FSM with support for terminal events:

Definition 29 *A terminating multi-transition finite state machine (abbreviated MT-FSM*) is a tuple (S, A, μ, μ^*) , where (S, A, μ) is an MT-FSM and*

μ^* is a map from $S - \{true, false\}$ to $MT(\{true, false\}, A)$. A terminating binary transition tree finite state machine (*BTT-FSM**) is a tuple (S, A, β, β^*) , where (S, A, β) is a *BTT-FSM* and β^* is a map from $S - \{true, false\}$ to $BTT(\{true, false\}, A)$. For a given event $\theta : A \rightarrow \{true, false\}$ in any of these finite state machines, we let $s \xrightarrow{\theta^*} true$ (or *false*) denote the fact that $\theta_{MT}(\mu^*(s)) = true$ (or *false*) or $\theta_{BTT}(\beta^*(s)) = true$ (or *false*), respectively. We call $s \xrightarrow{\theta^*} true$ (or *false*) a *terminating transition*. A sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n^*} true$ is called an *accepting sequence* (for s_1) and a sequence $s_1 \xrightarrow{\theta_1} s_2 \xrightarrow{\theta_2} \dots s_n \xrightarrow{\theta_n^*} false$ is called a *rejecting sequence* (for s_1). These notions also naturally extend to corresponding finite traces of events $e_{\theta_1} \dots e_{\theta_n}$.

Languages can be associated to states in *MT-FSM**s or *BTT-FSM**s as finite words of events.

Definition 30 Given a state $s \in S$ in an *MT-FSM** or in a *BTT-FSM** M , we let $\mathcal{L}_M(s)$ denote the set of finite traces $e_{\theta_1} \dots e_{\theta_n}$ with the property that $s \xrightarrow{\theta_1} \dots \xrightarrow{\theta_n^*} true$ is an accepting sequence in M . If a state $s_0 \in S$ is desired to be initial, then we write it at the end of the tuple, such as (S, A, μ, μ^*, s_0) or $(S, A, \beta, \beta^*, s_0)$, and let \mathcal{L}_M denote $\mathcal{L}_M(s_0)$. If $s_0 \xrightarrow{\theta_1} s_1 \dots \xrightarrow{\theta_n} s_n$ is a sequence of transitions from the initial state, then $e_{\theta_1} \dots e_{\theta_n}$ is called a *valid prefix* if and only if $e_{\theta_1} \dots e_{\theta_n} t \in \mathcal{L}_M$ for any (empty or not) trace t , and it is called an *invalid prefix* if and only if $e_{\theta_1} \dots e_{\theta_n} t \notin \mathcal{L}_M$ for any trace t .

The following is immediate.

Proposition 14 If $true \in S$ then $\mathcal{L}_M(true) = \mathcal{E}^*$, and if $false \in S$ then $\mathcal{L}_M(false) = \emptyset$. If $s \xrightarrow{\theta} s'$ in M then $\mathcal{L}_M(s') = \{t \mid e_{\theta}t \in \mathcal{L}_M(s)\}$; more generally, if $s \xrightarrow{\theta_1} s_1 \dots \xrightarrow{\theta_n} s_n$ is a sequence of transitions in M then $\mathcal{L}_M(s_n) = \{t \mid e_{\theta_1} \dots e_{\theta_n} t \in \mathcal{L}_M(s)\}$. In particular, if $s = s_0$ then $e_{\theta_1} \dots e_{\theta_n}$ is a *valid prefix* if and only if $\mathcal{L}_M(s_n) = \mathcal{E}^*$, and it is an *invalid prefix* if and only if $\mathcal{L}_M(s_n) = \emptyset$.

6.1.2 From MT-FSMs to BTT-FSMs

Supposing that one has encoded a logical requirement into an *MT-FSM* (we shall see how to do it for *LTL* in the next subsection), the next important step is to generate an efficient equivalent *BTT-FSM*. In the worst possible

case one just has to evaluate all the atomic predicates in order to proceed to the next state of a BTT-FSM, so they are preferred to MT-FSMs. What one needs to do is to develop a procedure that takes a multi-transition and returns a BTT. More BTTs can encode the same multi-transition, so one needs to develop some criteria to select the better ones. A natural selection criterion would be to minimize the average amount of computation. For example, if all atomic predicates are equally probable to hold and if an atomic predicate is very expensive to evaluate, then one would select that BTT that places the expensive predicate as deeply as possible, so its evaluation is delayed as much as possible. Based on the above, we believe that the following is an important theoretical problem in runtime monitoring:

Problem: Optimal BTT

Input: A set of atomic predicates a_1, \dots, a_k that hold with probabilities π_1, \dots, π_k and have costs c_1, \dots, c_k , respectively, and a multi-transition $p_1?s_1, \dots, p_n?s_n$ where p_1, \dots, p_n are Boolean formulae over a_1, \dots, a_k .

Output: A BTT implementing the multi-transition that probabilistically minimizes the amount of computation to decide the next state.

The probabilities and the costs in the problem above can be estimated either by static analysis of the source code of the program, or dynamically by first running and measuring the program several times, or by combinations of those. We do not know how to solve this interesting problem yet, but we conjecture the following result that we currently use in our implementation:

Conjecture 7 *If $\pi_1 = \dots = \pi_k$ and $c_1 = \dots = c_k$ then the solution to the problem above is the BTT of minimal size.*

Our current multi-transition to BTT algorithm is exponential in the number of atomic predicates; it simply enumerates all possible BTTs recursively and then selects the one of minimal size. Generating the BTTs takes significantly less time than generating the MT-FSM, so we do not regard it as a practical problem yet. However, it seems to be a challenging theoretical problem.

Example 6 *Consider again the traffic light controller safety property stating that “after green comes yellow”, which was written as $\square(\text{green} \rightarrow (!\text{red} \cup$*

State	MT for non-terminal events	MT for terminal events
1	<pre>[yellow \/ !green ? 1, !yellow /\ green /\ !red ? 2, !yellow /\ green /\ red ? false]</pre>	<pre>[yellow \/ !green ? true, !yellow /\ green ? false]</pre>
2	<pre>[yellow ? 1, !yellow /\ !red ? 2, !yellow /\ red ? false]</pre>	<pre>[yellow ? true, !yellow ? false]</pre>

Figure 6.2: MT-FSM for the formula $[\](\text{green} \rightarrow !\text{red} \cup \text{yellow})$.

`yellow`)) using LTL notation. Since more than one light can be lit at any moment, one should be very careful when expressing this safety property as an MT-FSM or a BTT-FSM.

Let us first express it as an MT-FSM. We need two states, one for the case in which green has not triggered yet the `!red U yellow` part and another for the case when it has. The condition to stay in state 1 is then `yellow \/ !green` and the condition to move to state 2 is `!yellow /\ green /\ !red`. If both a `green` and a `red` are seen then the machine should move to state false. The condition to move from state 2 back to state 1 is `yellow`, while the condition to stay in state 2 is `!yellow /\ !red`; `!yellow /\ red` should also move the machine in its false state. If the event is terminal then a `yellow` would make the reported answer true, i.e., the observed trace is an accepting sequence; if `yellow` is not true, then in state 1 the answer should be the opposite of `green`, while in state 2 the answer should be simply false. This MT-FSM is shown in Figure 6.2.

The interesting aspect of our FSMs is that not all the atomic predicates need to always be evaluated. For example, in state 2 of this MT-FSM the predicate `green` is not needed. A BTT-FSM further reduces the amount of predicates to be evaluated, by enforcing an “evaluate-by-need” policy. Figure 6.3 shows a BTT-FSM implementing the MT-FSM in Figure 6.2.

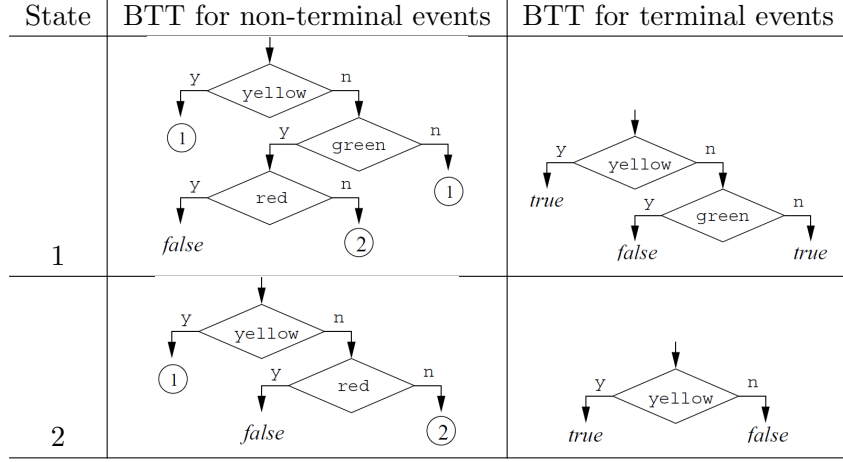


Figure 6.3: A BTT-FSM for the formula $\Box(\text{green} \rightarrow !\text{red} \cup \text{yellow})$.

6.2 Multi-Transitions and Binary Transition Trees

Büchi automata cannot be used unchanged as monitors. For the rest of the paper we explore structures suitable for monitoring as well as techniques to transform Büchi automata into such structures. Deterministic multi-transitions (*MT*) and binary-transition trees (*BTTs*) were introduced in [130, 224]. In this section we extend their original definitions with nondeterminism.

Definition 31 Let S and A be sets of **states** and **atomic predicates**, respectively, and let P_A denote the set of **propositions** over atoms in A , using the usual boolean operators. If $\{s_1, s_2, \dots, s_n\} \subseteq S$ and $\{p_1, p_2, \dots, p_n\} \subseteq P_A$, we call the n -tuple $[p_1 : s_1, p_2 : s_2, \dots, p_n : s_n]$ a (**nondeterministic**) **multi-transition (MT)** over P_A and S . Let $MT(P_A, S)$ denote the set of MTs over P_A and S .

Intuitively, if a monitor is in a state associated to an $MT[p_1 : s_1, p_2 : s_2, \dots, p_n : s_n]$ then p_1, p_2, \dots, p_n can be regarded as guards allowing the monitor to nondeterministically transit to one of the states s_1, s_2, \dots, s_n .

Definition 32 Maps $\theta : A \rightarrow \{\text{true}, \text{false}\}$ are called **A-events**, or simply **events**. Given an A-event θ , we define its **multi-transition extension** as the map $\theta_{MT} : MT(P_A, S) \rightarrow 2^S$, where $\theta_{MT}([p_1 : s_1, p_2 : s_2, \dots, p_n : s_n]) = \{s_i \mid \theta \models p_i\}$.

The role of A -events is to transmit the monitor information regarding the running program. In any program state, the map θ assigns atomic propositions to *true* iff they hold in that state, otherwise to *false*. Therefore, A -events can be regarded as abstractions of the program states. For an MT μ , the set of states $\theta_{MT}(\mu)$ is often called the set of *possible continuations* of μ under θ .

Example 7 If $\mu = [a \vee \neg b : s_1, \neg a \wedge b : s_2, c : s_3]$, and $\theta(a)=true$, $\theta(b)=false$, and $\theta(c)=true$, then the set of possible continuations of μ under θ , $\theta_{MT}(\mu)$, is $\{s_1, s_3\}$.

Definition 33 A (*nondeterministic*) *binary transition tree (BTT)* over A and S is inductively defined as either a set in 2^S or a structure of the form $a ? \beta_1 : \beta_2$, for some atom a and for some binary transition trees β_1 and β_2 . Let $BTT(A, S)$ denote the set of BTTs over the set of states S and atoms A .

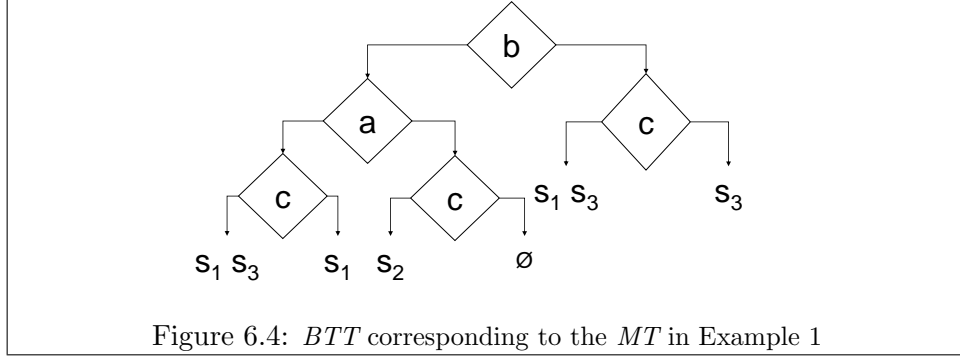
Definition 34 Given an event θ , we define its *binary transition tree extension* as the map $\theta_{BTT} : BTT(A, S) \rightarrow 2^S$, where:

$$\begin{aligned} \theta_{BTT}(Q) &= Q \text{ for any set of states } Q \subseteq S, \\ \theta_{BTT}(a ? \beta_1 : \beta_2) &= \theta_{BTT}(\beta_1) \text{ if } \theta(a) = true, \text{ and} \\ \theta_{BTT}(a ? \beta_1 : \beta_2) &= \theta_{BTT}(\beta_2) \text{ if } \theta(a) = false. \end{aligned}$$

Definition 35 A BTT β *implements* an MT μ , written $\beta \models \mu$, iff for any event θ , it is the case that $\theta_{BTT}(\beta) = \theta_{MT}(\mu)$.

Example 8 The BTT $b ? (a ? (c ? s_1 s_3 : s_1) : (c ? s_2 : \emptyset)) : (c ? s_1 s_3 : s_3)$ implements the multi-transition shown in Example 7.

Fig. 6.4 represents this *BTT* graphically. The right branch of the node labeled with **b** corresponds to the *BTT* expression $(c ? s_1 s_3 : s_3)$, and similarly for the left branch and every other node. Atomic predicates can be any host programming language boolean expressions. For example, one may be interested if a variable x is positive or if a vector $v[1...100]$ is sorted. Some atomic predicates typically are more expensive to evaluate than others. Since our purpose is to generate *efficient monitors*, we need to take the evaluation costs of atomic predicates into consideration. Moreover, some predicates can hold with higher probability than others; for example, some predicates may be simple “sanity checks”, such as checking whether the output of a sorting procedure is indeed sorted. We next assume that atomic predicates are given evaluation costs and probabilities to hold. These may be estimated *apriori*, either statically or dynamically.



Definition 36 If $\varsigma: A \rightarrow \mathcal{R}^+$ and $\pi: A \rightarrow [0, 1]$ are cost and probability functions for events in A , respectively, then let $\gamma_{\varsigma, \pi}: BTT(A, S) \rightarrow \mathcal{R}^+$ defined as:

$$\gamma_{\varsigma, \pi}(Q) = 0 \text{ for any } Q \subseteq S, \text{ and}$$

$$\gamma_{\varsigma, \pi}(a ? \beta_1 : \beta_2) = \varsigma(a) + \pi(a) * \gamma_{\varsigma, \pi}(\beta_1) + (1 - \pi(a)) * \gamma_{\varsigma, \pi}(\beta_2),$$

be the **expected (evaluation) cost** function on *BTTs* in $BTT(A, S)$.

Example 9 Given $\varsigma = \{(a, 10), (b, 5), (c, 20)\}$ and $\pi = \{(a, 0.2), (b, 0.5), (c, 0.5)\}$, the expected evaluation cost of the *BTT* defined in Example 2 is 30.

With the terminology and motivations above, the following problem develops as an interesting and important problem in monitor synthesis:

Problem: Optimal $BTT(A, S)$.

Input: A multi-transition $\mu = [p_1 : s_1, p_2 : s_2, \dots, p_n : s_n]$ with associated cost $\varsigma: A \rightarrow \mathcal{R}^+$ and probability $\pi: A \rightarrow [0, 1]$.

Output: A minimal cost *BTT* β with $\beta \models \mu$.

Binary decision trees (BDTs) and diagrams (BDDs) have been studied as models and data-structures for several problems in artificial intelligence [203] and program verification [68]. Appendix B discusses BDTs and how they relate to *BTTs*. Moret [203] shows that a simpler version of this problem, using BDTs, is NP-hard.

In spite of this result, in general the number of atoms in formulae is relatively small, so it is not impractical to exhaustively search for the optimal *BTT*. We next informally describe a backtracking algorithm that we are currently using in our implementation to compute the minimal cost *BTT* by exhaustive search. Start with the sequence of all atoms in A . Pick one atom, say a , and make two recursive calls to this procedure, each assuming one

boolean assignment to a . In each call, pass the remaining sequence of atoms to test, and simplify the set of propositions in the multi-transition according to the value of a . The product of the *BTTs* is taken when the recursive calls return in order to compute all *BTTs* starting with a . This procedure repeats until no atom is left in the sequence. We select the minimal cost *BTT* amongst all computed.

6.3 Binary Transition Tree Finite State Machines

We next define an automata-like structure, formalizing the desired concept of an *effective runtime monitor*. The transitions of each state are all-together encoded by a *BTT*, in practice the statistically optimal one, in order for the monitor to efficiently transit as events take place in the monitored program. Violations occur when one cannot further transit to any state for any event. A special state, called *neverViolate*, will denote a configuration in which one can no longer detect a violation, so one can stop the monitoring session if this state is reached.

Definition 37 A *binary transition tree finite state machine (BTT-FSM)* is a tuple $\langle A, S, btt, S_0 \rangle$, where A is a set of atoms, S is a set of states potentially including a special state called “*neverViolate*”, btt is a map associating a *BTT* in $BTT(A, S)$ to each state in S where $btt(\text{neverViolate}) = \{\text{neverViolate}\}$ when $\text{neverViolate} \in S$, and $S_0 \subseteq S$ is a subset of initial states.

Definition 38 Let $\langle A, S, btt, S_0 \rangle$ be a *BTT-FSM*. For an event θ and $Q, Q' \subseteq S$, we write $Q \xrightarrow{\theta} Q'$ and call it a **transition between sets of states**, whenever $Q' = \bigcup_{s \in Q} \theta_{BTT}(btt(s))$. A **trace of events** $\theta_1 \theta_2 \dots \theta_j$ generates a **sequence of transitions** $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ in the *BTT-FSM*, where $Q_0 = S_0$ and $Q_i \xrightarrow{\theta_{i+1}} Q_{i+1}$, for all $0 \leq i < j$. The trace is **rejecting** iff $Q_j = \{\}$.

Note that no finite extension of a trace $\theta_1 \theta_2 \dots \theta_j$ will be **rejected** if $\text{neverViolate} \in Q_j$. The state *neverViolate* denotes a configuration in which violations can no longer be detected for any finite trace extension. This means that the set Q_k will not be empty, for any $k > j$, when $\text{neverViolate} \in Q_j$. Therefore, the monitoring session can stop at event j if $\text{neverViolate} \in Q_j$, because we are only interested in violations of requirements.

Exercises

Exercise 8 *Give pseudo-code for the problem of generating a minimal BTT for a given MT (described in Section 6.1.2 and right after Definition 36).*

Exercise^{*} 9 *Prove or disprove Conjecture 7.*

Chapter 7

Monitor Synthesis: Extended Regular Expressions (ERE)

bad vs. good prefixes — cite Vardi

7.1 Monitoring ERE Safety Needs Non-Elementary Space

Extended regular expressions (EREs) add complementation (\neg) to regular expressions (REs). Complementation can be handy when defining safety properties, because it allows us to say both what should never happen as well as what could happen. In particular, in the context of monitoring EREs, one can switch between the expression of bad prefixes of a safety property and that of good prefixes by just applying a complement operator.

In this section we show that any monitor for safety properties expressed as EREs requires non-elementary space. More precisely, for a given $n \in \mathbb{N}$, we build an ERE of size $O(n^3)$ such that its language is prefix closed and any monitor for its language requires space

$$2^{2^{\cdot^{2^n}}} \quad \Bigg] \quad n \text{ nested power of 2 operations}$$

discuss that this also implies non-elementary time complexity

7.1.1 Discussion and Relevance of the Lower-Bound Result

move some of the discussion below to Chapter 6

Since regular expressions (REs) and deterministic (DFA) and non-deterministic (NFA) finite-state automata are enumerable structures while the set of safety properties is in bijection with the set of real numbers, there are many safety properties that cannot be expressed using REs or automata (or any other formalism whose objects are enumerable). Nevertheless, there are many safety properties of interest that can be expressed as REs or automata. Safety properties over finite or infinite traces can be expressed as REs in at least two different ways:

1. Use an RE to express the language of its bad prefixes; or
2. Use an RE to express the language of its (good) prefixes.

In the first case, the RE captures the finite-trace behaviors that should never happen, while in the second the RE captures the ones that could possibly happen.

Obviously, not all REs correspond to safety properties in one or the other of the two cases above. For example, the RE $(0 \cdot 1)^+$ cannot express the bad prefixes of a safety property, because “01” is a bad prefix while “010” is not. In order for an RE to express the bad prefixes of a safety property, it should have the property that once w is in its language, then all ww' for any w' should also be in its language. The RE $(0 \cdot 1)^+$ cannot express the good prefixes of a safety property either: “0101” is a good prefix while 010 is not. The language of an RE must be prefix closed in order to express the good prefixes of a safety property.

In both cases above, monitoring the safety property can be done very efficiently (linearly in the size of the RE, both space-wise and time-wise) by first translating its corresponding RE into an NFA, for example using a technique such as Thompson’s [262], and then doing one of the following:

1. In the first case, simulate the NFA-to-DFA construction on the fly as events are received. The state of the monitor is therefore a set of states of the NFA. At start, that set contains only the initial state of the NFA. For each new event, construct the next set by collecting all the NFA states that can be reached via the received event from any of the existing states in the set. If a final state is reached then report

violation of the property: bad prefix found. Since the final states in the NFA symbolize the reach of a bad prefix and since bad prefixes have “no future” in a safety property, the NFA associated to the original RA can be optimized (in case it is not already optimal directly from its construction) by removing any edges out of its final states.

2. In the second case, the monitor works the same way as in the first case, but checking at each time that the monitor state (also a set of NFA states) contains at least one final state of the NFA; if that is not the case, then report violation: prefix found which is not good. If one is willing to pay the exponential price and determinize the NFA of good prefixes, then one can further optimize the resulting DFA (in case it is not already optimized by construction) by collapsing all its non-final states into a “dead-end” state: indeed, the reachability of a non-final state signifies the reachability of a bad prefix, which has “no future”. It is not clear whether or how to optimize the NFA of good prefixes using the additional info that it is a safety property.

almost all the discussion above can go to Chapter 6.

Extended regular expressions (EREs) add complementation (\neg) to REs. Meyer and Stockmeyer [254, 255] showed that EREs can express languages non-elementarily more compactly than REs. In other words, for any constant $k \geq 1$, one can find EREs of large enough size $n \in \mathbb{N}$ for which there is no RE having the same language of size less than $2^{2^{\dots^{2^n}}}$, with k nested power operations. Meyer and Stockmeyer [254, 255] showed that several other problems concerning EREs are also non-elementary, including: the equivalence of EREs, the emptiness of the language of an ERE (and implicitly the emptiness of the complement of the language of an ERE), the automata generation (NFA or DFA) from an ERE, etc. Note that it is straightforward to generate potentially non-elementarily large automata from EREs. All one needs to do is to iteratively apply NFA-to-DFA transformations for EREs under complement operators and then complement the resulting DFAs (by complementing their final states). Since each NFA-to-DFA transformation may lead to an exponential explosion on the number of states and since the ERE can have arbitrarily many nested complement operators, the resulting NFA or DFA can be non-elementarily larger than the original ERE.

As already mentioned above, if we allow complementation then we can easily switch from an expression defining the bad prefixes of a safety property

to one defining its good prefixes, and backwards, by applying a complement operator. Therefore, from here on, when we say that an ERE expresses a safety property, without any loss of generality we assume that it defines the good prefixes of the safety property; in particular, we assume that its language is prefix closed. Clearly, if one can afford to generate an automaton from the ERE expressing a safety property, then one can and probably should use that automaton as a monitor for the safety property. However, since such an automaton can be enormous compared to the size of the original ERE, a natural question to ask is whether one can generate monitors for safety properties expressed as EREs that need less than non-elementary space in the size of the original ERE. We next give a negative answer to this question.

Let us first discuss our subsequent lower bound result from a more conceptual perspective. Notice that *synchronous monitoring*, that is, the monitoring process where an error is detected as soon as it appears, is harder than checking for satisfiability (or non-emptiness); indeed, if a formula in a particular formalism is not satisfiable (or it has an empty language), then a synchronous monitor should detect that before the first event is observed. We refer the interested reader to [223] for a discussion on various types of monitoring, including synchronous versus asynchronous monitoring. Synchronous monitors need to either directly (e.g., by accumulating logical constraints while verifying their consistency) or indirectly (e.g., by generating statically an automaton or a structure containing all possible future behaviors) check for satisfiability (or non-emptiness) of the remaining requirements as events are observed online. Unfortunately, this is an expensive process that may be desired to be avoided, at the expense of delaying the detection of violations. For example, the rewriting based monitoring approach for LTL in [223] delays the detection of violations of LTL formulae of the form “(next φ) and (next $\neg\varphi$)” for one more event, to avoid invoking an expensive satisfiability checker for LTL but to instead invoke a propositional satisfiability checker which is less expensive in practice; this is closely related to the notion of “informative prefixes” in [175] that “tell all the story”. Our subsequent lower-bound result states that any monitor for safety properties expressed using EREs, *synchronous or asynchronous*, requires non-elementary space.

Let us now clarify that our lower bound result is not a consequence of the lower-bound result by Meyer and Stockmeyer in [254] (see [255] for a proof of that result). A first reason is that neither the ERE constructed in [255, 254]

nor its complement is prefix closed. Indeed, we here focus on a subset of EREs, rather than arbitrary EREs, namely those whose languages are prefix closed, so they express good prefixes of safety properties. Supposing that one could modify the “hard” ERE in [255, 254] whose complement non-emptiness requires non-elementary space into one whose language is prefix-closed and whose size is linear in the size of the original one, the fact that synchronous monitoring of EREs is harder than checking for emptiness does not necessarily imply that monitoring that hard ERE requires non-elementary space. In fact, monitoring that particular ERE requires constant time to process each event, because it is equivalent with an automaton of one state – it takes, however, non-elementary space to compute that one state automaton. What it says is therefore that the *initialization step of ERE-safety synchronous monitoring* requires, in the worst case, non-elementary space.

Interestingly, if one could modify the results in [255, 254] to hold for prefix-closed EREs, including especially the result stating that finding an RE equivalent to an ERE is a non-elementary problem, then one could show that *synchronous monitoring* of safety properties expressed as EREs requires non-elementary space! Indeed, supposing that one had for any ERE a monitor that takes only elementary space in its worst-case monitoring scenario, then one could use that monitor to generate a DFA for the ERE as follows: start with the initial state of the monitor and then discover and store new states of the monitor by feeding arbitrary (but finite in number) events to each state of the monitor until no new state is discovered. This closure operation takes as much time and space as the number of states the monitor reach; since by assumption the monitor needs “only” elementary space to store its state in any scenario, we deduce that the obtained automaton has size elementary in the size of the ERE (and it also takes elementary time and space to generate it). In other words, we could find an elementary algorithm to associate to any (prefix closed) ERE an equivalent RE, contradicting the non-elementary lower bound in [255, 254] (again, supposing that the lower-bound results in [255, 254] could be modified for prefix-closed EREs).

Unfortunately, it is not that clear how to reduce the non-elementary problems in [255, 254] to *asynchronous monitoring*, and thus to conclude that asynchronous monitoring also requires non-elementary space. That is because a “smart” asynchronous monitor may in principle collapse states (when regarded as an automaton as in the construction above) in rather unexpected ways, just because it “knows” that eventually an error may be reported anyway if observation continues indefinitely from that state on; in

other words, states with the property that “eventually violation detected in the future” may be collapsed as equivalent by an asynchronous monitor. This way, the DFA extracted from a monitor for a safety property expressed as an ERE may be significantly smaller than the DFA corresponding to the ERE (and obviously, it may have a different language).

Our next result shows that asynchronous monitoring of ERE-safety also requires non-elementary space, which is a more general lower-bound result than the space non-elementarity of synchronous monitoring. Moreover, it gives an alternative proof of the non-elementary lower-bounds by Stockmeyer and Meyer [255, 254], because automata corresponding to safety-defining EREs are just special cases of monitors, so they must also take non-elementary space. Moreover, we improve the results in [255, 254] by showing that their lower-bounds also hold for a *subset of EREs*, namely those corresponding to safety properties.

Summarizing the discussion above, we believe that the main contributions of our subsequent lower-bound result are the following:

1. We show that asynchronous monitoring already requires non-elementary space, same as synchronous monitoring. For example, an ERE monitoring algorithm was presented by Roşu and Viswanathan in [219], which “rewrites” or “derives” the ERE by each letter in the input word; the derivation process consists of some straightforward rewrite rules, some for expanding the ERE others for contracting it via simplifications. No comprehensive and expensive check for emptiness on the resulting ERE is performed, except what is done by the simplification rules (for example $\emptyset \cdot R \rightarrow \emptyset$ and $\epsilon \cdot R \rightarrow R$, etc.). A check for emptiness can and should be eventually performed (for example, a check for emptiness can be done periodically, say every 10^x events for some convenient x). This gives us an asynchronous ERE monitoring algorithm, which, unlike the simplistic NFA/DFA generation algorithm, does not pay upfront the potentially non-elementary worst-case penalty! However, our subsequent lower bound result tells that there is a worst-case scenario in which one cannot avoid the non-elementary space required to store the continuously changing ERE if one wants to correctly eventually detect violations of the original ERE. And that is the case for any synchronous or asynchronous monitoring algorithm for safety properties expressed as EREs.
2. We propose a different technique to prove non-elementary lower-bounds, fundamentally different from the one in [255]. The technique in [255] is

based on diagonalization arguments and encodings of accepting Turing machine computations as finite trace words. Our technique is inspired from an idea by Chandra, Kozen and Stockmeyer [50] introduced to show the power of alternation and then used by several authors to prove exponential lower bounds [171, 172, 219, 223]. At our knowledge, the use of such a technique to show non-elementary lower bounds is novel. The idea of the technique in [50] is to define expressions having as languages $L_n = \{w^{(1)}\#w^{(2)}\#\dots\#w^{(k)}\$w \mid w^{(1)}, w^{(2)}, \dots, w^{(k)}, w \in \{0, 1\}^n, (\exists 1 \leq i \leq k) w^{(i)} = w\}$. Our idea is to define, using EREs, words of the form $X_n\$X'_n$, where X_n and X'_n are *n-deep nested sets* starting with elements in $\{0, 1\}^n$ (i.e., sets of sets of ... of sets of elements in $\{0, 1\}^n$, with n power set operations), such that X'_n is *n-nested included* in X_n , where *i-nested inclusion* is standard inclusion when $i = 1$ and, if $i > 1$, then it is defined inductively as: X'_i is *i-nested included* in X_i iff for each $X'_{i-1} \in X'_i$, there is some $X_{i-1} \in X_i$ such that X'_{i-1} is $(i - 1)$ -nested included in X_{i-1} .

One more observation is in place before we move on to the technical details. It is known that the *membership problem* for EREs, testing whether a word w of size m is in the language of an ERE of size n , is polynomial in m and n . For example, the classic algorithm by Hopcroft and Ullman [148] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$; slightly improved algorithms have been proposed by several authors [143, 269, 270, 271, 174, 157], reducing space requirements to $O(m^2 \cdot k + m \cdot n)$ and time to $O(m^3 \cdot k + m^2 \cdot n)$ or worse, where k is the number of complement operators in the ERE; a recent ERE membership algorithm was proposed by the author in [226], which runs in space $O(m \cdot \log m \cdot 2^n)$ and time $O(m^2 \cdot \log m \cdot 2^n)$ when $m > 2^n$. These algorithms appear to be efficient, because they are polynomial or simply exponential in the ERE, so one may think that one may device an elementary ERE-safety monitoring algorithm as follows: store the trace of events and at each newly received event invoke one of these “efficient” ERE membership algorithms. While this algorithm will indeed be elementary in the size of the ERE *and* the size of the trace, our lower bound result says that it will, in fact, be *non-elementary in the size of only the ERE!* In other words, for a carefully chosen “hard” ERE of size n , there are finite traces of large enough size m so that checking their membership is a problem which is non-elementary in n ; this will indeed happen when m is non-elementarily larger than n .

7.1.2 The Lower-Bound Result

EREs define languages by inductively applying *union* ($+$), *concatenation* (\cdot), *Kleene Closure* (\star), *intersection* (\cap), and *complementation* (\neg). The language of an ERE R , say $\mathcal{L}(R)$, is defined inductively as follows, where $s \in \Sigma$:

- $\mathcal{L}(\emptyset) = \emptyset$,
- $\mathcal{L}(\epsilon) = \{\epsilon\}$,
- $\mathcal{L}(s) = \{s\}$,
- $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
- $\mathcal{L}(R_1 \cdot R_2) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$,
- $\mathcal{L}(R^\star) = (\mathcal{L}(R))^\star$,
- $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$,
- $\mathcal{L}(\neg R) = \neg \mathcal{L}(R)$.

If R does not contain \neg and \cap then it is a *regular expression* (RE). By applying De Morgan's law $R_1 \cap R_2 \equiv \neg(\neg R_1 + \neg R_2)$, EREs can be linearly (in both time and size) translated into equivalent EREs without intersection; therefore, intersection of EREs is just syntactic sugar. The *size* of an ERE is the total number of occurrences of letters and composition operators ($+$, \cdot , \star , and \neg) that it contains. In what follows we assume that Σ is finite. For notational simplicity, in what follows we let Σ also denote the RE $s_1 + s_2 + \dots + s_n$ where $\Sigma = \{s_1, s_2, \dots, s_n\}$ and let Σ^\star denote both the language $\{s_1, s_2, \dots, s_n\}^\star$ and the RE $(s_1 + s_2 + \dots + s_n)^\star$.

For $n \in \mathbb{N}$, let us define inductively the following alphabets and languages:

- $\Sigma_0 = \{0, 1\}$ and $\Psi_0 = \{0, 1\}^n$, and
- $\Sigma_i = \Sigma_{i-1} \cup \{\#_i\}$ and $\Psi_i = \{\#_i \#_i\} \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\}$, for all $1 \leq i \leq n$.

In the above, $\#_i$ are n fresh letters. The intuition for the languages Ψ_i above is to encode nested sets of depth i that contain sets of words of n bits at their deepest level. The symbols $\#_i$ play the role of markers separating the elements of such sets. For example, the word $\#_2 \#_1 \#_1 \#_2 \#_1 01 \#_1 10 \#_1 \#_2 \#_1 \#_1 \#_2$

encodes $\{\{\}, \{01, 10\}, \{\}\}$, that is, the set $\{\{\}, \{01, 10\}\}$; since the multiplicity and order of elements in sets are irrelevant, the same set can have (infinitely) many different encodings. Formally, let us define the following *set* functions associating to encodings in Ψ_i corresponding nested sets:

- $set_0 : \Psi_0 \rightarrow \{0, 1\}^n$ is the identity function on Ψ_0 , i.e., $set_0(w) = w$;
- $set_i : \Psi_i \rightarrow \mathcal{P}^i(\{0, 1\}^n)$, where \mathcal{P}^i is the power set operator applied i times, $set_i(\#_i \#_i) = \{\}$, $set_i(\#_i X_{i-1} \#_i) = \{set_{i-1}(X_{i-1})\}$, and $set_i(\#_i X_{i-1} X_i) = \{set_{i-1}(X_{i-1})\} \cup set_i(X_i)$, for all $1 \leq i \leq n$, $X_{i-1} \in \Psi_{i-1}$, and $X_i \in \Psi_i$.

Note that $|set_0(\Psi_0)| = 2^n$ and $set_i(\Psi_i) = \mathcal{P}(set_{i-1}(\Psi_{i-1}))$ for all $1 \leq i \leq n$; therefore, $|set_i(\Psi_i)| = 2^{2^{i-1} \cdot 2^n}$ for all $1 \leq i \leq n$, with $i + 1$ nested power operations.

Let us define *nested-inclusion* relations $_{-} \subseteq_i _{-} : \mathcal{P}^i(\{0, 1\}^n) \times \mathcal{P}^i(\{0, 1\}^n)$ for $0 \leq i \leq n$ and *nested-membership* relations $_{-} \in_i _{-} : \mathcal{P}^{i-1}(\{0, 1\}^n) \times \mathcal{P}^i(\{0, 1\}^n)$ for $1 \leq i \leq n$ as follows:

- $_{-} \subseteq_0 _{-}$ is the identity on $\{0, 1\}^n$ and $_{-} \subseteq_1 _{-}$ is $_{-} \subseteq _{-} : \mathcal{P}(\{0, 1\}^n) \times \mathcal{P}(\{0, 1\}^n)$,
- $_{-} \in_1 _{-}$ is $_{-} \in _{-} : \{0, 1\}^n \times \mathcal{P}(\{0, 1\}^n)$, and for $1 < i$,
- $S_{i-1} \in_i S_i$ iff there is some $S'_{i-1} \in S_i$ such that $S_{i-1} \subseteq S'_{i-1}$, and
- $S_i \subseteq_i S'_i$ iff $S_{i-1} \in_i S'_i$ for each $S_{i-1} \in S_i$.

For example, if $n = 2$ then $\{\{00, 01\}, \{01, 10\}, \{11\}\} \subseteq_2 \{\{00, 01, 10\}, \{00, 11\}\}$ because $\{00, 01\}, \{01, 10\} \subseteq_1 \{00, 01, 10\}$ and $\{11\} \subseteq_1 \{00, 11\}$.

We can now define Σ as $\Sigma_n \cup \{\$\}$ and the infinite trace property P_n^ω over Σ :

$$(\epsilon \cup (\Sigma_n^* \cup \{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \subseteq_n set_n(X_n)\}) \cdot \{\$\}) \cdot \Sigma_n^\omega.$$

An infinite trace in P_n^ω therefore contains at most two $\$$ letters and infinitely many letters in Σ_n . There are no restrictions on the appearance of the letters in Σ_n when there is no $\$$ letter or when there is precisely one $\$$ letter. However, if the infinite trace contains precisely two $\$$ letters, that is, if it has the form $w \$ w' \$ u$ for some $w, w' \in \Sigma_n^*$ and some $u \in \Sigma_n^\omega$, then w and w' must be in Ψ_n and the nested set corresponding to w must nested-include the nested set corresponding to w' ; there are no restrictions on u .

Proposition 15 $P_n^\omega \in \text{Safety}^\omega$.

Proof: There are two cases to analyze for an infinite trace that is not in P_n^ω : when it contains more than two \$ letter, or when it has the form $w\$w'\u with $w, w' \in \Sigma_n^*$ and $u \in \Sigma_n^\omega$, but it is not the case that $w, w' \in \Psi_n$ and $\text{set}_n(w') \subseteq_n \text{set}_n(w)$. In the first case, we can pick the first prefix of the infinite trace containing three \$ letters in total; clearly, this finite trace prefix cannot be continued into any acceptable infinite trace. In the second case, since there are no restrictions on the letters in u , we can easily see that the prefix $w\$w'\$$ is already a violation threshold: there is no $u' \in \Sigma_n^\omega$ such that $w\$w'\$u' \in P_n^\omega$. \square

The bijection in the proof of Theorem 3 associates to each infinite-trace safety property a persistent finite-trace safety property by taking its prefixes. Let P_n be the persistent finite-trace safety property $\text{prefixes}(P_n^\omega)$ corresponding to P_n^ω . It is easy to see that P_n is the property

$$\Sigma_n^* \cup \Sigma_n^* \cdot \{\$ \} \cdot \Sigma_n^* \cup \{X_n\$X'_n \mid X_n, X'_n \in \Psi_n \text{ and } \text{set}_n(X'_n) \subseteq_n \text{set}_n(X_n)\} \cdot \{\$ \} \cdot \Sigma_n^*.$$

Note that monitoring P_n^ω is the same as monitoring P_n : in both cases, besides the capability to checking whether there are more than two \$ letters, which is trivial, the monitor needs to store sufficient information about the nested set corresponding to X_n , so that, when the first \$ is seen, to be able to check whether it nested-includes the set corresponding to the upcoming, yet unknown X'_n .

Theorem 8 *Any synchronous or asynchronous monitor for P_n or P_n^ω needs space non-elementary in n , namely $\Omega(2^{2^{\cdot^{2^n}}})$, with n nested power operations.*

Proof: Suppose that M is a monitor for P_n or P_n^ω and suppose that, during a monitoring session, it reads the prefix $X_n \in \Psi_n$. Regardless of how M is defined or implemented, in particular regardless of whether it reports violations synchronously or asynchronously, when the first \$ event is encountered, the state of M must contain enough information to sooner or later be able to decide whether the set $\text{set}_n(X'_n)$ corresponding to *any* upcoming (unknown at the time the \$ is observed) sequence X'_n is nested-included in $\text{set}_n(X_n)$. Since $\text{set}_n(X'_n)$ can in particular be equal to $\text{set}_n(X_n)$, and since once the second \$ event is observed there is no further event that may bring new knowledge to the monitor, we deduce that M must be able

to distinguish any two different sets in $set_n(\Psi_n)$ when the first \$ event is encountered, that is, M 's states must be different after reading words in Ψ_n whose corresponding nested sets are different. Therefore, M must be able to distinguish $|set_n(\Psi_n)|$ different possibilities. Since one needs $\Omega(\log N)$ space to distinguish among N different situations (one label for each), we conclude that M needs space $\Omega(\log(|set_n(\Psi_n)|))$, that is, $\Omega(2^{2^n})$ with n nested power operations. Hence, any monitor for P_n needs non-elementarily large space in n . \square

We next show how to construct an ERE polynomial in size with n whose language is precisely P_n .

Theorem 9 *There is an ERE of size $O(n^3)$ whose language is P_n .*

Proof: Note that P_n is a union of three languages, the first two being trivial to express as languages of corresponding REs. As expected, the difficult part is to associate an ERE to the language

$$\{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \subseteq_n set_n(X_n)\}.$$

Note that so far we did not need complementation. The property above can, however, be expressed as an ERE of size $O(n^3)$ using $O(n)$ nested complement operators. The idea is to define iteratively a sequence of EREs \mathbb{K}_i for $0 \leq i \leq n$ whose languages contain words $X_i w \$ w' X'_i$ with $set_i(X'_i) \subseteq_i set(X_i)$, which are contiguous fragments of desired words $X_n \$ X'_n$. Then \mathbb{K}_n would be the language that we are looking for. To define \mathbb{K}_i , we observe that the nested-inclusion $S'_i \subseteq_i S_i$ is equivalent to: there is no $S'_{i-1} \in S'_i$ such that it is not the case that we can find some $S_{i-1} \in S_i$ such that $S'_{i-1} \subseteq_{i-1} S_{i-1}$. This crucial observation will allow us to define \mathbb{K}_i in terms of \mathbb{K}_{i-1} . We next develop the technical details.

Let us first define regular patterns corresponding to each of the languages Ψ_i for $0 \leq i \leq n$; to avoid introducing new names, we ambiguously let the corresponding regular expressions have the same names as their languages:

- Let $\Psi_0 = (0 + 1)^n$, where for an RE, r^n is $r \cdot r \cdot \dots \cdot r$ (n times); and
- Let $\Psi_i = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^+ \cdot \#_i$ for all $1 \leq i \leq n$.

Iteratively eliminating the Ψ_{i-1} regular expressions from the right-hand-sides, we eventually obtain $n + 1$ regular patters, each of size $O(n)$ (the size of Ψ_0 as a regular expression is $O(n)$ and each Ψ_i adds a constant size to that of Ψ_{i-1}).

Next we define REs for the languages $\text{prefixes}(\Psi_i)$ and $\text{suffixes}(\Psi_i)$ of prefixes and respectively suffixes of words in Ψ_i , for all $0 \leq i \leq n$. We only discuss the prefix closure languages; the suffix closures are dual. The prefix closures can be defined relatively easily inductively as follows:

- $\text{prefixes}(\Psi_0) = \bigcup_{k=0}^n \{0, 1\}^k$, and
- $\text{prefixes}(\Psi_i) = \{\epsilon, \#_i \#_i\} \cup \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1}) \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1})$
 $= \{\#_i \#_i\} \cup (\{\#_i\} \cdot \Psi_{i-1})^* \cdot (\{\epsilon\} \cup \{\#_i\} \cdot \text{prefixes}(\Psi_{i-1}))$.

These languages can be expressed with the following REs; as before, we ambiguously use the same names for the corresponding REs:

- Let $\text{prefixes}(\Psi_0) = \epsilon + (0 + 1) + (0 + 1)^2 + \dots + (0 + 1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\text{prefixes}(\Psi_i) = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^* \cdot (\epsilon + \#_i \cdot \text{prefixes}(\Psi_{i-1}))$.

Iteratively eliminating the REs $\text{prefixes}(\Psi_{i-1})$ from the right-hand-sides, we eventually obtain $n + 1$ REs, each of size $O(i^2 + n)$ (the size of $\text{prefixes}(\Psi_0)$ as an RE is $O(n)$ and each $\text{prefixes}(\Psi_i)$ adds size $O(i)$ to that of $\text{prefixes}(\Psi_{i-1})$). Dually,

- Let $\text{suffixes}(\Psi_0) = \epsilon + (0 + 1) + (0 + 1)^2 + \dots + (0 + 1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\text{suffixes}(\Psi_i) = \#_i \cdot \#_i + (\epsilon + \text{suffixes}(\Psi_{i-1}) \cdot \#_i) \cdot (\Psi_{i-1} \cdot \#_i)^*$.

We next define REs L_i and R_i for $0 \leq i \leq n$ whose languages contain the contiguous fragments of words in Ψ_n that are allowed to appear to the left and to the right of $\$,$ respectively, so that words in $\mathcal{L}(L_i)$ start with $\#_i$ and words in $\mathcal{L}(R_i)$ end with $\#_i$. Let us also assume by convention that $L_{n+1} = R_{n+1} = \epsilon$ (the RE whose language contains only the empty word). It is easy to see that L_i and R_i can be defined as follows:

- Let $L_i = \#_i \cdot \Sigma_n^* \cap \text{suffixes}(\Psi_n)$, and
- Let $R_i = \Sigma_n^* \cdot \#_i \cap \text{prefixes}(\Psi_n)$.

Note that words in L_i and R_i may not necessarily start or end with a word in Ψ_i : indeed, the $\#_i$ that may start or end L_i or R_i could very well be followed or preceded, respectively, by a $\#_{i+1}$ or, if $i = n$, by $\$$.

Let us also define the EREs \bar{L}_i and \bar{R}_i whose languages are included in those of L_i and R_i , respectively, and whose words start or end with words in Ψ_i :

- Let $\bar{L}_i = \Psi_i \cdot \Sigma_n^* \cap \text{suffixes}(\Psi_n)$, and
- Let $\bar{R}_i = \Sigma_n^* \cdot \Psi_i \cap \text{prefixes}(\Psi_n)$.

It is not difficult to see that $\bar{L}_i = \Psi_i \cdot L_{i+1}$ and $\bar{R}_i = R_{i+1} \cdot \Psi_i$. Note that the sizes of L_i , R_i , \bar{L}_i and \bar{R}_i are $O(n^2)$.

Let us now define the EREs \mathbb{K}_i for $0 \leq i \leq n$ as follows:

- $\mathbb{K}_0 = \bar{L}_0 \cdot \$ \cdot \bar{R}_0 \cap \bigcap_{k=0}^{n-1} (\Sigma_0^k \cdot 0 \cdot \Sigma^\star \cdot 0 \cdot \Sigma_0^{n-k-1} + \Sigma_0^k \cdot 1 \cdot \Sigma^\star \cdot 1 \cdot \Sigma_0^{n-k-1})$,
and
- $\mathbb{K}_i = \bar{L}_i \cdot \$ \cdot \bar{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i)$.

We next show by induction on i that $\mathcal{L}(\mathbb{K}_i)$ is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \subseteq_i \text{set}_i(X_i)\}.$$

It is easy to see that $\mathcal{L}(\mathbb{K}_0) = \{X_0 w X_0 \mid X_0 \in \Psi_0, w \in \mathcal{L}(L_1 \cdot \$ \cdot R_1)$, because the large conjunct in \mathbb{K}_0 states that the words formed with the first n letters and with the last ones, respectively, are equal and in Ψ_0 , and because $\bar{L}_0 \cdot \$ \cdot \bar{R}_0 = \Psi_0 \cdot L_1 \cdot \$ \cdot R_1 \cdot \Psi_0$ and \subseteq_0 is the identity on $\{0, 1\}^n$. For the inductive step, let us now assume that for some arbitrary $1 \leq i < n$, $\mathcal{L}(\mathbb{K}_{i-1})$ is the language

$$\{X_{i-1} w X'_{i-1} \mid X_{i-1}, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_i \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \subseteq_{i-1} \text{set}_{i-1}(X_{i-1})\}.$$

Then we can easily show that $\mathcal{L}((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1})$ is the language

$$\{X_i w X'_{i-1} \mid X_i \in \Psi_i, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \subseteq_{i-1} \text{set}_i(X_i)\}.$$

Then we can show that $\mathcal{L}(\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \bar{L}_i \cdot \$ \cdot \bar{R}_{i-1})$ is

$$\{X_i w X'_{i-1} \mid X_i \in \Psi_i, X'_{i-1} \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), \text{set}_{i-1}(X'_{i-1}) \not\subseteq_{i-1} \text{set}_i(X_i)\}.$$

Now we can show that $\mathcal{L}(\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \bar{L}_i \cdot \$ \cdot \bar{R}_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i$ is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \not\subseteq_{i-1} \text{set}_i(X_i)\}.$$

Finally, we are now able to show that $\mathcal{L}(\mathbb{K}_i)$, that is,

$$\mathcal{L}(\overline{L}_i \cdot \$ \cdot \overline{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^* \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^* \cdot \#_i))$$

is the language

$$\{X_i w X'_i \mid X_i, X'_i \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), \text{set}_i(X'_i) \subseteq_i \text{set}_i(X_i)\}.$$

Since $L_{n+1} = R_{n+1} = \epsilon$, it follows that

$$\mathcal{L}(\mathbb{K}_n) = \{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n, \text{set}_i(X'_i) \subseteq_i \text{set}_i(X_i)\}.$$

The size of \mathbb{K}_n is $O(n^3)$.

We can now show that the language of the ERE of size $O(n^3)$

$$(\epsilon + (\Sigma_n^* + \mathbb{K}_n) \cdot \$) \cdot \Sigma_n^*$$

is indeed P_n . □

We can now formulate our space lower-bound result for monitoring ERE-safety as a corollary of the two results above.

Corollary 1 *For any $n \in \mathbb{N}$, there is some safety property whose good prefixes are precisely the words in the language of an ERE of size $O(n)$ and whose monitoring (synchronous or asynchronous) requires space $\Omega(2^{2^{\frac{2}{3}\sqrt{n}}})$ with $\sqrt[3]{n}$ nested power operations.*

7.2 Generating Optimal Monitors for ERE

incorporate RTA'03 paper with Mahesh

Abstract: Software engineers and programmers can easily understand regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must not occur during an execution. Complementation gives one the power to express patterns on strings more compactly. In this paper we present a technique to generate optimal monitors from EREs. Our monitors are deterministic finite automata (DFA) and our novel contribution is to generate them using a modern coalgebraic technique called coinduction. Based on experiments with our implementation, which can be publicly tested and used over the web, we believe that our technique is more efficient than the simplistic method based on complementation of automata which can quickly lead to a highly-exponential state explosion.

7.2.1 Introduction

Regular expressions can express patterns in strings in a compact way. They proved very useful in practice; many programming/scripting languages like Perl, Python, Tcl/Tk support regular expressions as core features. Because of their power to express a rich class of patterns, regular expressions, are used not only in computer science but also in various other fields, such as molecular biology [167]. All these applications boast of very efficient implementation of regular expression pattern matching and/or membership algorithms. Moreover, it has been found that compactness of regular expressions can be increased non-elementarily by adding complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition (R^*) operators of regular expressions. These are known as *extended regular expressions* (EREs) and they proved very intuitive and succinct in expressing regular patterns.

Recent trends have shown that the software analysis community is inclining towards scalable techniques for software verification. Works in [128] merged temporal logics with testing, hereby getting the benefits of both

worlds. The Temporal Rover tool (TR) and its follower DB Rover [87] are already commercial. In these tools the Java code is instrumented automatically so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool [165, 184] has been developed to monitor safety properties in interval past time temporal logics. In [209, 214], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [125] is a runtime verification environment currently under development at NASA Ames. The Java MultiPathExplorer tool [239] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. [111, 127] present efficient algorithms for monitoring future time temporal logic formulae, while [130] gives a technique to synthesize efficient monitors from past time temporal formulae. [219] uses rewriting to perform runtime monitoring of EREs.

An interesting aspect of EREs is that they can express safety properties compactly, like those encountered in testing and monitoring. By generating automata from logical formulae, several of the works above show that the safety properties expressed by different variants of temporal logics are subclasses of regular languages. The converse is *not* true, because there are regular patterns which cannot be expressed using temporal logics, most notoriously those related to counting; e.g., the regular expression $(0 \cdot (0+1))^*$ saying that every other letter is 0 does not admit an equivalent temporal logic formula. Additionally, EREs tend to be often very natural and intuitive in expressing requirements. For example, let us try to capture the safety property “it should not be the case that in any trace of a traffic light we see green and then immediately red at any point”. The natural and intuitive way to express it in ERE is $\neg((-\emptyset) \cdot \text{green} \cdot \text{red} \cdot (-\emptyset))$, where \emptyset is the empty ERE (no words), so $-\emptyset$ means “anything”.

Previous approaches to ERE membership testing [143, 205, 269, 174, 158] have focussed on developing techniques that are polynomial in both the size of the word and the size of the formulae. The best known result in these approaches is described in [174] where they can check if a word satisfies an ERE in time $O(m \cdot n^2)$ and space $O(m \cdot m + k \cdot n^2)$, where m is the size of the ERE, n is the length of the word, and k is the number of negation/intersection operators. These algorithms, unfortunately, cannot be used for the purpose of monitoring. This is because they are not incremental. They assume the entire word is available before their execution. Additionally, their running time and space requirements are quadratic in the size of the trace. This

is unacceptable when one has a long trace of events and wants to monitor a small ERE, as it is typically the case. This problem is removed in [219] where traces are checked against EREs through incremental rewriting. At present, we do not know if the technique in [219] is optimal or not.

A simple, straightforward, and practical approach is to generate optimal *deterministic finite automata* (DFA) from EREs [147]. This process involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom-up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential.

Our approach is to generate the minimal DFA from an ERE using coinductive techniques. In this paper, the DFA thus generated is called the *optimal monitor* for the given ERE. Currently, we are not aware of any other algorithm that does this conversion in a straightforward way. The complexity of our algorithm seems to be hard to evaluate, because it depends on the size of the minimal DFA associated to an ERE and we are not aware of any lower bound results in this direction. However, experiments are very encouraging. Our implementation, which is available for evaluation on the internet via a CGI server reachable from <http://fs1.cs.uiuc.edu/rv/>, rarely took longer than one second to generate a DFA, and it took only 18 minutes to generate the minimal 107 state DFA for the ERE in Example 16 which was used to show the exponential space lower bound of ERE monitoring in [219].

In a nutshell, in our approach we use the concept of derivatives of an ERE, as described in Subsection 7.2.2. For a given ERE one generates all possible derivatives of the ERE for all possible sequences of events. The size of this set of derivatives depends upon the size of the initial ERE. However, several of these derivative EREs can be equivalent to each other. One can check the equivalence of EREs using coinductive technique as described in Section 7.2.3, that generates a set of equivalent EREs, called *circularities*. In Section 7.2.5, we show how circularities can be used to construct an efficient algorithm that generates optimal DFAs from EREs. In Section 16.7, we describe an implementation of this algorithm and give performance analysis

results. We also made available on the internet a CGI interface to this algorithm.

7.2.2 Extended Regular Expressions and Derivatives

In this section we recall extended regular expressions and their derivatives.

Extended Regular Expressions

Extended regular expressions (ERE) define languages by inductively applying union (+), concatenation (\cdot), Kleene Closure (*), intersection (\cap), and complementation (\neg). More precisely, for an alphabet E , whose elements are called *events* in this paper, an ERE over E is defined as follows, where $a \in E$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \cap R \mid \neg R.$$

The language defined by an expression R , denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset, \\ \mathcal{L}(\epsilon) &= \{\epsilon\}, \\ \mathcal{L}(A) &= \{A\}, \\ \mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\ \mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\ \mathcal{L}(R^*) &= (\mathcal{L}(R))^*, \\ \mathcal{L}(R_1 \cap R_2) &= \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\ \mathcal{L}(\neg R) &= \Sigma^* \setminus \mathcal{L}(R). \end{aligned}$$

Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. The translation only results in a linear blowup in size. Therefore, in the rest of the paper we do not consider expressions containing intersection. More precisely, we only consider EREs of the form

$$R ::= R + R \mid R \cdot R \mid R^* \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

Derivatives

In this subsection we recall the notion of *derivative*, or “residual” (see [14, 13], where several interesting properties of derivatives are also presented). It is based on the idea of “event consumption”, in the sense that an extended regular expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w , $aw \in R$ if and only if $w \in R\{a\}$.

In the rest of the paper assume defined the typical operators on EREs and consider that the operator $+$ is associative and commutative and that the operator \cdot is associative. In other words, reasoning is performed modulo the equations:

$$\begin{aligned} (R_1 + R_2) + R_3 &= R_1 + (R_2 + R_3), \\ R_1 + R_2 &= R_2 + R_1, \\ (R_1 \cdot R_2) \cdot R_3 &= R_1 \cdot (R_2 \cdot R_3). \end{aligned}$$

We next consider an operation $_ \{ _ \}$ which takes an ERE and an event, and give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$\begin{aligned} (R_1 + R_2)\{a\} &= R_1\{a\} + R_2\{a\} & (1) \\ (R_1 \cdot R_2)\{a\} &= (R_1\{a\}) \cdot R_2 + \text{if } (\epsilon \in R_1) \text{ then } R_2\{a\} \text{ else } \emptyset \text{ fi} & (2) \\ (R^*)\{a\} &= (R\{a\}) \cdot R^* & (3) \\ (\neg R)\{a\} &= \neg(R\{a\}) & (4) \\ b\{a\} &= \text{if } (b == a) \text{ then } \epsilon \text{ else } \emptyset \text{ fi} & (5) \\ \epsilon\{a\} &= \emptyset & (6) \\ \emptyset\{a\} &= \emptyset & (7) \end{aligned}$$

The right-hand sides of these equations use operations which we describe next. “if $(_)$ then $_$ else $_$ fi” takes a boolean term and two EREs as arguments and has the expected meaning defined by two equations:

$$\begin{aligned} \text{if } (true) \text{ then } R_1 \text{ else } R_2 \text{ fi} &= R_1 & (8) \\ \text{if } (false) \text{ then } R_1 \text{ else } R_2 \text{ fi} &= R_2 & (9) \end{aligned}$$

We assume a set of equations that properly define boolean expressions and reasoning. Boolean expressions include the constants *true* and *false*, as well as the usual connectors $_ \wedge _$, $_ \vee _$, and *not*. Testing for empty trace membership (which is used by (2)) can be defined via the following equations:

$$\epsilon \in (R_1 + R_2) = (\epsilon \in R_1) \vee (\epsilon \in R_2) \quad (10)$$

$$\epsilon \in (R_1 \cdot R_2) = (\epsilon \in R_1) \wedge (\epsilon \in R_2) \quad (11)$$

$$\epsilon \in (R^*) = \text{true} \quad (12)$$

$$\epsilon \in (\neg R) = \text{not}(\epsilon \in R) \quad (13)$$

$$\epsilon \in b = \text{false} \quad (14)$$

$$\epsilon \in \epsilon = \text{true} \quad (15)$$

$$\epsilon \in \emptyset = \text{false} \quad (16)$$

The 16 equations above are natural and intuitive. [219] shows that these equations, when regarded as rewriting rules are terminating and ground Church-Rosser (modulo associativity and commutativity of $_+ _$ and modulo associativity of $_ \cdot _$), so they can be used as a functional procedure to calculate derivatives. Due to the fact that the 16 equations defining the derivatives can generate useless terms, in order to keep EREs compact we also propose defining several *simplifying equations*, including at least the following:

$$\emptyset + R = R,$$

$$\emptyset \cdot R = \emptyset,$$

$$\epsilon \cdot R = R,$$

$$R + R = R.$$

The following result (see, e.g., [219] for a proof) gives a simple procedure, based on derivatives, to test whether a word belongs to the language of an ERE:

Theorem 10 *For any ERE \mathcal{R} and any events a, a_1, a_2, \dots, a_n in A , the following hold:*

1. $a_1 a_2 \dots a_n \in \mathcal{L}(R\{a\})$ if and only if $aa_1 a_2 \dots a_n \in \mathcal{L}(R)$; and

2. $a_1 a_2 \dots a_n \in \mathcal{L}(R)$ if and only if $\epsilon \in R\{a_1\}\{a_2\}\dots\{a_n\}$.

7.2.3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [215, 113, 114], to test whether two EREs are equivalent, that is, if they have the same language. Since the goal of this paper is to translate an ERE into a minimal DFA, standard techniques for checking equivalence, such as translating the two expressions into DFAs and then comparing those, do not make sense in this framework.

A particularly appealing aspect of circular coinduction in the framework of EREs is that it does not only show that two EREs are equivalent, but also generates a larger set of equivalent EREs which will all be used in order to generate the target DFA.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. To keep the presentation simple and self contained, in this section we define an oversimplified version of hidden logic together with its associated circular coinduction proof rule, still general enough to support defining and proving EREs behaviorally equivalent.

Algebraic Preliminaries

The reader is assumed familiar with basic equational logic and algebra in this section. We recall a few notions in order to just make our notational conventions precise. An S -sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ -algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S -sorted signature Σ and an S -indexed set of variables Z , let $T_\Sigma(Z)$ denote the Σ -term algebra over variables in Z . If $V \subseteq S$ then $\Sigma|_V$ is a V -sorted signature consisting of all those operations in Σ with sorts entirely in V . We may let $\sigma(X)$ denote the term $\sigma(x_1, \dots, x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with only variables as arguments except one, and we do not care which one, which is t . If t is a Σ -term of sort s' over a special variable $*$ of sort s and A is a Σ -algebra, then $A_t : A_s \rightarrow A_{s'}$ is the usual interpretation of t in A .

7.2.4 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H) -signature, say Σ , is a many sorted $(V \cup H)$ -signature. A *hidden subsignature* of Σ is a hidden (V, H) -signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$. The *data signature* is $\Sigma|_V$. An operation of visible result not in $\Sigma|_V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden

signature Σ with a fixed subsignature Γ . Informally, Σ -algebras are universes of possible states of a system, i.e., “black boxes,” where one is only concerned with behavior under experiments with operations in Γ , where an experiment is an observation of a system attribute after perturbation; this is formalized below.

A Γ -context for sort $s \in V \cup H$ is a term in $T_\Gamma(\{ * : s \})$ with one occurrence of $*$. A Γ -context of visible result sort is called a Γ -experiment. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ -algebra A with a hidden subsignature Γ , for sorts $s \in (V \cup H)$, we define Γ -behavioral equivalence of $a, a' \in A_s$ by $a \equiv_\Sigma^\Gamma a'$ iff $A_c(a) = A_c(a')$ for all Γ -experiments c ; we may write \equiv instead of \equiv_Σ^Γ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with \equiv_Σ^Γ . Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ -behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ .

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ -algebra A Γ -behaviorally satisfies a Σ -equation $(\forall X) t = t'$, say e , iff for each $\theta : X \rightarrow A$, $\theta(t) \equiv_\Sigma^\Gamma \theta(t')$; in this case we write $A \models_\Sigma^\Gamma e$. If E is a set of Σ -equations we then write $A \models_\Sigma^\Gamma E$ when A Γ -behaviorally satisfies each Σ -equation in E . We may omit Σ and/or Γ from \models_Σ^Γ when they are clear.

A behavioral Σ -specification is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ , and E is a set of Σ -sentences equations. Non-data Γ -operations (i.e., in $\Gamma - \Sigma|_V$) are called *behavioral*. A Σ -algebra A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models_\Sigma^\Gamma E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models_\Sigma^\Gamma e$.

EREs can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for equivalence of EREs. [113] shows how standard regular expressions (without negation) can be defined as a behavioral specification, a BOBJ implementation, and also how BOBJ with its circular coinductive rewriting algorithm can prove automatically several equivalences of regular expressions. Related interesting work can also be found in [230]. In this paper we extend that to general EREs,

generate minimal observer monitors, and also give several other examples.

Example 10 *A behavioral specification of EREs defines a set of two visible sorts $V = \{Bool, Event\}$, one hidden sort $H = \{Ere\}$, one behavioral attribute $\epsilon \in _ : Ere \rightarrow Bool$ and one behavioral method, the derivative, $_ \{-\} : Ere \times Event \rightarrow Ere$, together with all the other operations in Subsection 7.2.2 defining EREs, including the events in E which are defined as visible constants of sort $Event$, and all the equations in Subsection 7.2.2. We call it the ERE behavioral specification and let \mathcal{B}_{ERE} denote it.*

*Since the only behavioral operators are the test for ϵ membership and the derivative, it follows that the experiments have exactly the form $\epsilon \in * \{a_1\} \{a_2\} \dots \{a_n\}$, for any events a_1, a_2, \dots, a_n . In other words, an experiment consists of a series of derivations followed by an ϵ membership test, and therefore two regular expressions are behavioral equivalent if and only if they cannot be distinguished by such experiments. Notice that the above reasoning applies within any algebra satisfying the presented behavioral specification. The one we are interested in is, of course, the free one, whose set carriers contain exactly the extended regular expressions as presented in Subsection 7.2.2, and the operations have the obvious interpretations. We informally call it the ERE algebra.*

Letting \equiv denote the behavioral equivalence relation generated on the ERE algebra, then Theorem 10 immediately yields the following important result.

Theorem 11 *If R_1 and R_2 are two EREs then $R_1 \equiv R_2$ if and only if $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.*

This theorem allows us to prove equivalence of EREs by making use of behavioral inference in the ERE behavioral specification, from now on simply referred to by \mathcal{B} , including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show EREs equivalent.

Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [215] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed

as a very powerful automated technique to show behavioral equivalence. We let \Vdash denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before we present circular coinduction formally, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ . Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some “frozen” instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [114]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b , and for each (hidden or visible) sort s in the specification, a special operation $[-] : s \rightarrow b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification \mathcal{B} and any equation $(\forall X) t = t'$, it is necessarily the case that $\mathcal{B} \Vdash (\forall X) t = t'$ iff $\mathcal{B} \Vdash (\forall X) [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X) [t] = [t']\} \Vdash (\forall X, W) [\delta(t, W)] = [\delta(t', W)], \text{ for all appropriate } \delta \in \Gamma}{\mathcal{B} \Vdash (\forall X) t = t'}$$

We call the equation $(\forall X) [t] = [t']$ added to \mathcal{B} a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 12 *The usual equational inference rules together with Circular Coinduction are sound. That means that if $\mathcal{B} \Vdash (\forall X) t = t'$ and $\text{sort}(t, t') \neq b$, or if $\mathcal{B} \Vdash (\forall X) [t] = [t']$, then $\mathcal{B} \models (\forall X) t = t'$.*

Example 11 *Suppose that we want to show that the EREs $(a + b)^*$ and $(a^*b^*)^*$ admit the same language. By Theorem 11, we can instead show that $\mathcal{B}_{ERE} \models (\forall \emptyset) (a + b)^* = (a^*b^*)^*$. Notice that a and b are treated as constant events here; one can also prove the result when a and b are variables, but one would need to first make use of the theorem of hidden constants [215]. To simplify writing, we omit the empty quantifier of equations. By the Circular Coinduction rule, one generates the following three proof obligations*

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [\epsilon \in (a + b)^*] = [\epsilon \in (a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [(a^*b^*)^*\{b\}]. \end{aligned}$$

The first proof task follows immediately by using the equations in \mathcal{B} as rewriting rules, while the other two tasks reduce to

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*]. \end{aligned}$$

By applying Circular Coinduction twice, after simplifying the two obvious proof tasks stating the ϵ membership, one gets the following four proof obligations

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [a^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [a^*(a^*b^*)^*\{b\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{a\}] = [b^*(a^*b^*)^*\{a\}], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*\{b\}] = [b^*(a^*b^*)^*\{b\}], \end{aligned}$$

which, after simplification translate into

$$\begin{aligned} \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [a^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [a^*(a^*b^*)^*], \\ \mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^*b^*)^*], [(a + b)^*] = [b^*(a^*b^*)^*]\} &\Vdash [(a + b)^*] = [b^*(a^*b^*)^*], \end{aligned}$$

Again by applying circular coinduction we get

$$\begin{array}{l}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash
\end{array}$$

which now follow all immediately. Notice that *BOBJ* uses the newly added (to \mathcal{B}_{ERE}) equations as rewriting rules when it applies its circular coinductive rewriting algorithm, so the proof above is done slightly differently, but entirely automatically.

Example 12 Suppose now that one wants to show that $\neg(a^*b) \equiv \epsilon + a^* + (a+b)^*b(a+b)(a+b)^*$. One can also do it entirely automatically by circular coinduction as above, generating the following list of circularities:

$$\begin{array}{lcl}
[\neg(a^*b)] & = & [\epsilon + a^* + (a+b)^*b(a+b)(a+b)^*], \\
[\neg(\epsilon)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^* + (a+b)^*].
\end{array}$$

Example 13 One can also show by circular coinduction that concrete EREs satisfy systems of guarded equations. This is an interesting but unrelated subject, so we do not discuss it in depth here. However, we show how easily one can prove by coinduction that a^*b is the solution of the equation $R = a \cdot R + b$. This equation can be given by adding a new ERE constant r to \mathcal{B}_{ERE} , together with the equations $\epsilon \in r = \text{false}$, $r\{a\} = r$, and $r\{b\} = \epsilon$. Circular Coinduction applied on the goal $r = a^*b$ generates the proof tasks:

$$\begin{array}{lcl}
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [\epsilon \in r] = [\epsilon \in a^*b], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{a\}] = [a^*b\{a\}], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{b\}] = [a^*b\{b\}],
\end{array}$$

which all follow immediately.

The following says that circular coinduction provides a decision procedure for equivalence of EREs.

Theorem 13 *If R_1 and R_2 are two EREs, then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if $\mathcal{B}_{ERE} \Vdash R_1 = R_2$. Moreover, since the rules in \mathcal{B}_{ERE} are ground Church-Rosser and terminating, circular coinductive rewriting [113, 114], which iteratively rewrites proof tasks to their normal forms followed by a one step coinduction if needed, gives a decision procedure for ERE equivalence.*

7.2.5 Generating Minimal DFA Monitors by Coinduction

In this section we show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate a minimal DFA from any ERE. This DFA can then be used as an optimal monitor for that ERE. The main idea here is to associate states in DFA to EREs obtained by deriving the initial ERE; when a new ERE is generated, it is tested for equivalence with all the other already generated EREs by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that, once an equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent equivalent EREs. These can be used to later quickly infer the other equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for EREs, which is described below.

We are given an initial ERE R_0 over alphabet A and from that we want to generate the equivalent minimal DFA $D = (S, A, \delta, s_0, F)$, where S is the set of states, $\delta : S \times A \rightarrow S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of final states. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove equivalence of two regular expressions in the algorithm. The algorithm maintains the set of states S in the form of non-equivalent EREs. At the beginning of the algorithm S is initialized with a single element, which is the given ERE R_0 . Next, we generate all the derivatives of the initial ERE one by one in a depth first manner. A derivative $R_x = R\{x\}$ is added to the set S , if the set does not contain any ERE equivalent to the derivative R_x . We then extend the transition function by setting $\delta(R, x) = R_x$. If an ERE R' equivalent to the

derivative already exists in the set S , we extend the transition function by setting $\delta(R, x) = R'$. To check if an ERE equivalent to the derivative R_x already exists in the set S , we sequentially go through all the elements of the set S and try to prove its equivalence with R_x . In testing the equivalence we first add the set of circularities to the initial \mathcal{B} . Then we invoke the coinductive procedure. If for some ERE $R' \in S$, we are able to prove that $R' \equiv R_x$ i.e. $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash R' = R_x$, then we add the new equivalences Eq_{new} , created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven equivalences in future proofs.

The derivatives of the initial ERE R_0 with respect to all events in the alphabet A are generated in a depth first fashion. The pseudo code for the whole algorithm is given in Figure 1.

```

dfs( $R$ )
begin
  foreach  $x \in A$  do
     $R_x \leftarrow R\{x\}$ ;
    if  $\exists R' \in S$  such that  $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash R' = R_x$  then
       $\delta(R, x) = R'$ ;  $Eq_{\text{all}} \leftarrow Eq_{\text{all}} \cup Eq_{\text{new}}$ 
    else  $S \leftarrow S \cup \{R_x\}$ ;  $\delta(R, x) = R_x$ ; dfs( $R_x$ ); fi
  endfor
end

```

Figure 7.1: ERE to minimal DFA generation algorithm

In the procedure **dfs** the set of final states F consists of the EREs from S which contain ϵ . This can be tested efficiently using the equations (10-16) in Subsection 7.2.2. The DFA generated by the procedure **dfs** may now contain some states which are non-final and from which the DFA can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the DFA.

Theorem 14 *If D is the DFA generated for a given ERE R by the above algorithm then*

1. $\mathcal{L}(D) = \mathcal{L}(R)$,
2. D is the minimal DFA accepting $\mathcal{L}(R)$.

Proof: Suppose $a_1a_2 \dots a_n \in \mathcal{L}(R)$. Then $\epsilon \in R\{a_1\}\{a_2\} \dots \{a_n\}$. If $R_i = R\{a_1\}\{a_2\} \dots \{a_i\}$ then $R_{i+1} = R_i\{a_{i+1}\}$. To prove that $a_1a_2 \dots a_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $R_i \equiv \delta(R, a_1a_2 \dots a_i)$. For the base case if $R_1 \equiv R\{a_1\}$ then **dfs** extends the transition function by setting $\delta(R, a_1) = R$. Therefore, $R_1 \equiv R = \delta(R, a_1)$. If $R_1 \not\equiv R$ then **dfs** extends δ by setting $\delta(R, a_1) = R_1$. So $R_1 \equiv \delta(R, a_1)$ holds in this case also. For the induction step let us assume that $R_i \equiv R' = \delta(R, a_1a_2 \dots a_i)$. If $\delta(R', a_{i+1}) = R''$ then from the **dfs** procedure we can see that $R'' \equiv R'\{a_{i+1}\}$. However, $R_i\{a_{i+1}\} \equiv R'\{a_{i+1}\}$. So $R_{i+1} \equiv R'' = \delta(R', a_{i+1}) = \delta(R, a_1a_2 \dots a_{i+1})$. Also notice $\epsilon \text{ in } R_n \equiv \delta(R, a_1a_2 \dots a_n)$; this implies that $\delta(R, a_1a_2 \dots a_n)$ is a final state and hence $a_1a_2 \dots a_n \in \mathcal{L}(D)$.

Now suppose $a_1a_2 \dots a_n \in \mathcal{L}(D)$. The proof that $a_1a_2 \dots a_n \in \mathcal{L}(R)$ goes in a similar way by showing that $R_i \equiv \delta(R, a_1, a_2 \dots a_i)$. \square

7.2.6 Implementation and Evaluation

We have implemented the coinductive rewriting engine in the rewriting specification language Maude 2.0 [70]. The interested readers can download the implementation from the website <http://fsl.cs.uiuc.edu/rv/>. The operations on extended regular languages that are supported by our implementation are \sim for negation, $_*$ for Kleene Closure, $_ _$ for concatenation, $_ \& _$ for intersection, and $_ + _$ for union in increasing order of precedence. Here, the intersection operator $_ \& _$ is a syntactic sugar and is translated to an ERE containing union and negation using De Morgan's Law:

$$\text{eq } R1 \ \& \ R2 = \sim (\sim R1 + \sim R2) \ .$$

To evaluate the performance of the algorithm we have generated the minimal DFA for all possible EREs of size up to 9. Surprisingly, the size of any DFA for EREs of size up to 9 did not exceed 9. Here the number of states gives the size of a DFA. The following table shows the performance of our procedure for the worst EREs of a given size. The code is executed on a

Pentium 4 2.4GHz, 4 GB RAM linux machine.

Size	ERE	no. of states in DFA	Time (ms)	Rewrites
4	$\neg (a \ b)$	4	< 1	863
5	$(a \neg b)^*$	4	< 1	1370
6	$\neg ((a \neg b)^*)$	4	1	1453
7	$\neg (a \neg a \ a)$	6	1	2261
8	$\neg ((a \neg b)^* \ b)$	7	1	3778
9	$\neg (a \neg a \ b) \ b$	9	5	9717

Example 14 In particular, for the ERE $\neg (a \neg a \ b) \ b$ the generated minimal DFA is given in Figure 7.2.

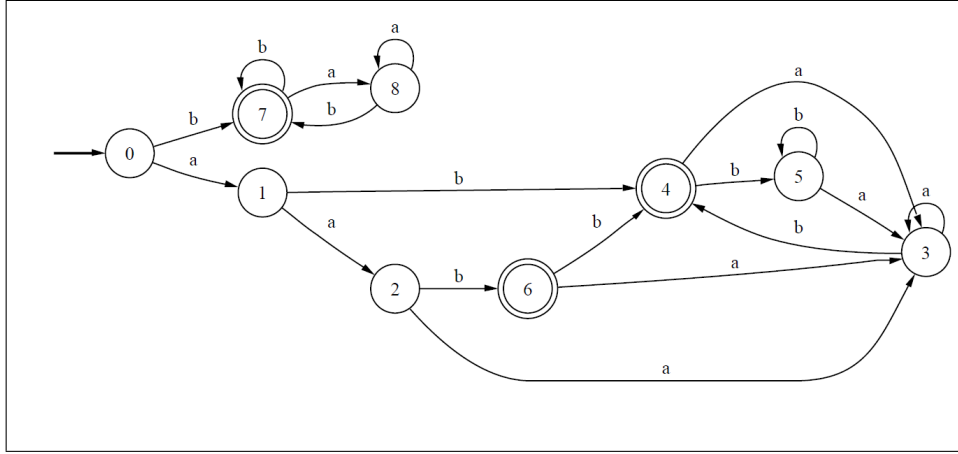


Figure 7.2: $\neg (a \neg a \ b) \ b$

Example 15 The ERE $\neg ((\neg \text{empty}) \text{ (green red)} (\neg \text{empty}))$ states the safety property that it should not be the case that in any trace of a traffic light we see green and red consecutively at any point. The set of events are assumed to be $\{\text{green, red, yellow}\}$. We think that this is the most intuitive and natural expression for this safety property. The implementation took 1ms and 1663 rewrites to generate the minimal DFA with 2 states. The DFA is given in Figure 7.3.

However for large EREs the algorithm may take a long time to generate a minimal DFA. The size of the generated DFA may grow non-elementarily

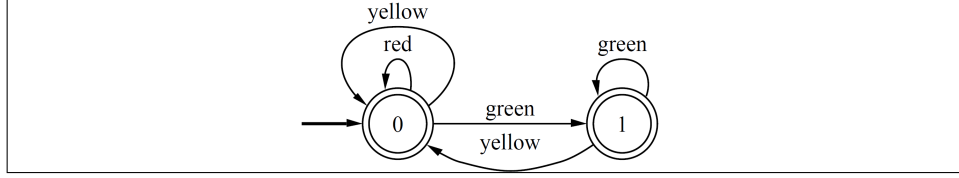


Figure 7.3: $\neg ((\neg \text{empty}) (\text{green red}) (\neg \text{empty}))$

in the worst case. We generated DFAs for some complex EREs of larger sizes and got relatively promising results. One such sample result is as follows.

Example 16 *Let us consider the following ERE of size 110*

$$\begin{aligned}
 &(\neg \$)^* \$ (\neg \$)^* \cap \\
 &\quad (0 + 1 + \#)^* \# (\\
 &\quad \quad ((0 + 1) 0 \# (0 + 1 + \#)^* \$ (0 + 1) 0 + (0 + 1) 1 \# (0 + 1 + \#)^* \$ (0 + 1) 1) \\
 &\quad \quad \cap (0(0 + 1) \# (0 + 1 + \#)^* \$ 0(0 + 1) + 1(0 + 1) \# (0 + 1 + \#)^* \$ 1(0 + 1))) .
 \end{aligned}$$

This ERE accepts the language L_2 , where

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}$$

The language L_k was first introduced in [49] to show the power of alternation, used in [219] to show an exponential lower bound on ERE monitoring, and in [171, 172] to show the lower bounds for model checking. Our implementation took almost 18 minutes to generate the minimal DFA of size 107 and in the process it performed 1,374,089,220 rewrites.

The above example shows that the procedure can take a large amount of time and space to generate DFAs for large EREs. To avoid the computation associated with the generation of minimal DFA we plan to maintain a database of EREs and their corresponding minimal DFAs on the internet. Whenever someone wants to generate the minimal DFA for a given ERE he/she can look up the internet database for the minimal DFA. If the ERE and the corresponding DFA exists in the database he/she can retrieve the corresponding DFA and use it as a monitor. Otherwise, he/she can generate the minimal DFA for the ERE and submit it to the internet database to create a new entry. The database will check the equivalence of the submitted ERE and the corresponding minimal DFA and insert it in the database. In this way one can avoid the computation of generating minimal DFA if it is already done by someone else. To further reduce the computation, circularities could also be stored in the database.

Online Monitor Generation and Visualization

We have extended our implementation to create an internet server for optimal monitor generation that can be accessed from the url <http://fsl.cs.uiuc.edu/rv/>. Given an ERE the server generates the optimal DFA monitor for a user. The user submits the ERE through a web based form. A CGI script handling the web form takes the submitted ERE as an input, invokes the Maude implementation to generate the minimal DFA, and presents it to the user either as a graphical or a textual representation. To generate the graphical representation of the DFA we are currently using the GraphViz tool [105].

We presented a new technique to generate optimal monitors for extended regular expressions, which avoids the traditional technique based on complementation of automata, that we think is quite complex and not necessary. Instead, we have considered the (co)algebraic definition of EREs and applied coinductive inferencing techniques in an innovative way to generate the minimal DFA. Our approach to store already proven equivalences has resulted into a very efficient and straightforward algorithm to generate minimal DFA. We have evaluated our implementation on several hundreds EREs and have got promising results in terms of running time. Finally we have installed a server on the internet which can generate the optimal DFA for a given ERE.

At least two major contributions have been made. Firstly, we have shown that coinduction is a viable and quite practical method to prove equivalence of extended regular expressions. Previously this was done only for regular expressions without complementation. Secondly, building on the coinductive technique, we have devised an algorithm to generate minimal DFAs from EREs. At present we have no bound for the size of the optimal DFA, but we know for sure that the DFAs we generate are indeed optimal. However we know that the size of an optimal DFA is bounded by some exponential in the size of the ERE. As future work, it seems interesting to investigate the size of minimal DFAs generated from EREs, and also to apply our coinductive techniques to generate monitors for other logics, such as temporal logics.

Exercises

Exercise 10 *Does the lower bound result in Section 7.1 hold for semi-extended regular expressions, that is, for regular expressions extended only with intersection (\cap) but not negation (\neg)? If yes, then sketch a proof. If no, then explain why not and try to find another lower bound.*

Exercise 11 *Implement the derivative-based ERE membership checking algorithm described in Section 7.2.2 (see also Theorem 10). Use your favorite programming language.*

Exercise* 12 *Implement the coinductive monitor synthesis for ERE described in Section 7.2.5.*

Chapter 8

Monitor Synthesis: Linear Temporal Logic (LTL)

add co-inductive algorithm

8.1 Finite Trace Future Time Linear Temporal Logic

Classical (infinite trace) linear temporal logic (LTL) [212, 186, 188] provides in addition to the propositional logic operators the temporal operators \Box (always), \Diamond (eventually), \cup (until), and \circ (next). An LTL standard model is a function $t : \mathcal{N}^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions \mathcal{P} , i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a natural number) into the set of propositions that hold at that point. The operators have the following interpretation on such an infinite trace. Assume formulae x and y . The formula $\Box x$ holds if x holds in all time points, while $\Diamond x$ holds if x holds in some future time point. The formula $x \cup y$ (x until y) holds if y holds in some future time point, and until then x holds (so we consider strict until). Finally, $\circ x$ holds for a trace if x holds in the suffix trace starting in the next (the second) time point. The propositional operators have their obvious meaning. For example, the formula $\Box (x \rightarrow \Diamond y)$ is true if for any time point (\Box) it holds that if x is true then eventually (\Diamond) y is true. It is standard to define a core LTL using only atomic propositions, the propositional operators \neg (not) and \wedge (and), and the temporal operators \circ and \cup , and then define all other propositional and temporal operators as derived constructs. Standard equations are $\Diamond x = \text{true} \cup x$ and $\Box x =$

!<>!X.

Our goal is to develop a framework for testing software systems using temporal logic. Tests are performed on finite execution traces and we therefore need to formalize what it means for a *finite trace* to satisfy an LTL formula. We first present a semantics of finite trace LTL using standard mathematical notation. Then we present a specification in Maude of a finite trace semantics. Whereas the former semantics uses universal and existential quantification, the second Maude specification is defined using recursive definitions that have a straightforward operational rewriting interpretation and which therefore can be executed.

8.1.1 Finite Trace Semantics

As mentioned in Subsection 2.1.2, a trace is viewed as a non-empty finite sequence of program states, each state denoting the set of propositions that hold at that state. We shall first outline the finite trace LTL semantics using standard mathematical notation rather than Maude notation. The debatable issue here is what happens at the end of the trace. The choice to validate or invalidate all the atomic propositions does not work in practice, because there might be propositions whose values are always opposite to each other, such as, for example, “gate up” and “gate down”. Driven by experiments, we found that a more reasonable assumption is to regard a finite trace as an infinite stationary trace in which the last event is repeated infinitely.

Assume two total functions on traces, $head : \text{Trace} \rightarrow \text{Event}$ returning the head event of a trace and $length$ returning the length of a finite trace, and a partial function $tail : \text{Trace} \rightarrow \text{Trace}$ for taking the tail of a trace. That is, $head(e, t) = head(e) = e$, $tail(e, t) = t$, and $length(e) = 1$ and $length(e, t) = 1 + length(t)$. Assume further for any trace t , that t_i denotes the suffix trace that starts at position i , with positions starting at 1. The satisfaction relation $\models \subseteq \text{Trace} \times \text{Formula}$ defines when a trace t satisfies a formula f , written $t \models f$, and is defined inductively over the structure of the formulae as follows, where A is any atomic proposition and X and Y are any formulae:

$t \models \text{true}$	iff	true ,
$t \models \text{false}$	iff	false ,
$t \models A$	iff	$A \in \text{head}(t)$,
$t \models X \wedge Y$	iff	$t \models X$ and $t \models Y$,
$t \models X ++ Y$	iff	$t \models X$ xor $t \models Y$,
$t \models \circ X$	iff	(if $\text{tail}(t)$ is defined then $\text{tail}(t) \models X$ else $t \models X$),
$t \models <>X$	iff	$(\exists i \leq \text{length}(t)) t_i \models X$,
$t \models []X$	iff	$(\forall i \leq \text{length}(t)) t_i \models X$,
$t \models X \cup Y$	iff	$(\exists i \leq \text{length}(t)) (t_i \models Y \text{ and } (\forall j < i) t_j \models X)$.

The semantics of the “next” operator reflects perhaps best the stationarity assumption of last events in finite traces.

Notice that finite trace LTL can behave quite differently from standard infinite trace LTL. For example, there are formulae which are not valid in infinite trace LTL but valid in finite trace LTL, such as $<>([]A \wedge []!A)$ for any atomic proposition A , and there are formulae which are satisfiable in infinite trace LTL and not satisfiable in finite trace LTL, such as the negation of the above. The formula above is satisfied by any finite trace because the last event/state in the trace either contains A or it does not.

8.1.2 Finite Trace Semantics in Maude

Now it can be relatively easily seen that the following Maude specification correctly “defines” the finite trace semantics of LTL described above. The only important deviation from the rigorous mathematical formulation described above is that the quantifiers over finite sets of indexes are expressed recursively.

```
fmod LTL is
  extending PROP-CALC .
*** syntax
  op []_ : Formula -> Formula [prec 11] .
  op <>_ : Formula -> Formula [prec 11] .
  op _U_ : Formula Formula -> Formula [prec 14] .
  op o_ : Formula -> Formula [prec 11] .
*** semantics
  vars X Y : Formula .
  var E : Event .
  var T : Trace .
  eq E |= o X = E |= X .
  eq E,T |= o X = T |= X .
```

```

eq E    |= <> X  = E |= X .
eq E,T  |= <> X  = E,T |= X or T |= <> X .
eq E    |= [] X  = E |= X .
eq E,T  |= [] X  = E,T |= X and T |= [] X .
eq E    |= X U Y = E |= Y .
eq E,T  |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
endfm

```

Notice that only the temporal operators needed declarations and semantics, the others being already defined in PROP-CALC and LOGICS-BASIC, and that the definitions that involved the functions *head* and *tail* were replaced by two alternative equations.

One can now directly verify LTL properties on finite traces using Maude's rewriting engine. Consider as an example a traffic light that switches between the colors *green*, *yellow*, and *red*. The LTL property that after *green* comes *yellow*, and its negation, can now be verified on a finite trace using Maude's rewriting engine, by typing joining commands to Maude such as:

```

reduce green, yellow, red, green, yellow, red, green, yellow, red, red
      |= [] (green -> !red U yellow) .
reduce green, yellow, red, green, yellow, red, green, yellow, red, red
      |= !([] (green -> !red U yellow)) .

```

which should return the expected answers, i.e., **true** and **false**, respectively. The algorithm above does nothing but blindly follows the mathematical definition of satisfaction and even runs reasonably fast for relatively small traces. For example, it takes¹ about 30ms (74k rewrite steps) to reduce the first formula above and less than 1s (254k rewrite steps) to reduce the second on traces of 100 events (10 times larger than the above). Unfortunately, this algorithm does not seem to be tractable for large event traces, even if run on very fast platforms. As a concrete practical example, it took Maude 7.3 million rewriting steps (3 seconds) to reduce the first formula above and 2.4 billion steps (1000 seconds) for the second on traces of 1,000 events; it could not finish in one night (more than 10 hours) the reduction of the second formula on a trace of 10,000 events. Since the event traces generated by an executing program can easily be larger than 10,000 events, the trivial algorithm above cannot be used in practice.

A rigorous complexity analysis of the algorithm above is hard (because it has to take into consideration the evaluation strategy used by Maude for

¹On a 1.7GHz, 1Gb memory PC.

terms of sort `Bool`) and not worth the effort. However, a simplified worse-case analysis can be easily made if one only counts the maximum number of atoms of the form `event` \models `atom` that can occur during the rewriting of a satisfaction term, as if all the Boolean reductions were applied after all the other reductions: let us consider a formula $X = \Box(\Box(\dots(\Box A)\dots))$ where the always operator is nested m times, and a trace T of size n , and let $T(n, m)$ be the total number of basic satisfactions `event` \models `atom` that occur in the normal form of the term $T \models X$ if no Boolean reductions were applied. Then, the recurrence formula $T(n, m) = T(n - 1, m) + T(n, m - 1)$ follows immediately from the specification above. Since $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$, it follows that $T(n, m) > \binom{n}{m}$, that is, $T(n, m) = \Omega(n^m)$, which is of course unacceptable.

8.2 A Backwards, Asynchronous, but Efficient Algorithm

The satisfaction relation above for finite trace LTL can hence be defined recursively, both on the structure of the formulae and on the size of the execution trace. As is often the case for functions defined this way, an efficient *dynamic programming* algorithm can be generated from any LTL formula. We first show how such an algorithm looks for a particular formula, and then present the main algorithm generator. The work in this section appeared as a technical report [222], but for a slightly different finite trace LTL, namely one in which all the atomic propositions were considered *false* at the end of the trace. As explained previously in the paper, we are now in the favor of a semantics where traces are considered stationary in their last event. The generated dynamic programming algorithms are as efficient as they can be and one can hope: linear in both the trace and the LTL formula. Unfortunately, they need to traverse the execution trace backwards, so they are trace storing and asynchronous. However, a similar but dual technique applies to past time LTL, producing very efficient forwards and synchronous algorithms [130, 129].

8.2.1 An Example

The formula we choose below is artificial (and will not be used later in the paper), but contains all four temporal operators. We believe that this example would practically be sufficient for the reader to foresee the general algorithm

presented in the remaining of the section. Let $\Box((p \cup q) \rightarrow \langle \rangle (q \rightarrow or))$ be an LTL formula and let $\varphi_1, \varphi_2, \dots, \varphi_{10}$ be its subformulae, in breadth-first order:

$$\begin{aligned}
\varphi_1 &= \Box((p \cup q) \rightarrow \langle \rangle (q \rightarrow or)), \\
\varphi_2 &= (p \cup q) \rightarrow \langle \rangle (q \rightarrow or), \\
\varphi_3 &= p \cup q, \\
\varphi_4 &= \langle \rangle (q \rightarrow or), \\
\varphi_5 &= p, \\
\varphi_6 &= q, \\
\varphi_7 &= q \rightarrow or, \\
\varphi_8 &= q, \\
\varphi_9 &= or, \\
\varphi_{10} &= r.
\end{aligned}$$

Given any finite trace $t = e_1 e_2 \dots e_n$ of n events, one can recursively define a matrix $s[1..n, 1..10]$ of Boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$ as follows:

$$\begin{aligned}
s[i, 10] &= (r \in e_i) \\
s[i, 9] &= s[i + 1, 10] \\
s[i, 8] &= (q \in e_i) \\
s[i, 7] &= s[i, 8] \text{ implies } s[i, 9] \\
s[i, 6] &= (q \in e_i) \\
s[i, 5] &= (p \in e_i) \\
s[i, 4] &= s[i, 7] \text{ or } s[i + 1, 4] \\
s[i, 3] &= s[i, 6] \text{ or } (s[i, 5] \text{ and } s[i + 1, 3]) \\
s[i, 2] &= s[i, 3] \text{ implies } s[i, 4] \\
s[i, 1] &= s[i, 2] \text{ and } s[i + 1, 1],
\end{aligned}$$

for all $i < n$, where *and*, *or*, *implies* are ordinary Boolean operations and $==$ is the equality predicate, where $s[n, 1..10]$ are defined as below:

$$\begin{aligned}
s[n, 10] &= (r \in e_n) \\
s[n, 9] &= s[n, 10] \\
s[n, 8] &= (q \in e_n) \\
s[n, 7] &= s[n, 8] \text{ implies } s[n, 9] \\
s[n, 6] &= (q \in e_n) \\
s[n, 5] &= (p \in e_n) \\
s[n, 4] &= s[n, 7] \\
s[n, 3] &= s[n, 6] \\
s[n, 2] &= s[n, 3] \text{ implies } s[n, 4] \\
s[n, 1] &= s[n, 2].
\end{aligned}$$

Note again that the trace needs to be *traversed backwards*, and that the row n of s is filled according to the stationary view of finite traces in their last event. An important observation is that, like in many other dynamic programming algorithms, one does not have to store all the table $s[1..n, 1..10]$, which would be quite large in practice; in this case, one needs only two rows, $s[i, 1..10]$ and $s[i + 1, 1..10]$, which we shall write *now* and *next* from now on, respectively. It is now only a simple exercise to write up the following algorithm:

```

INPUT: trace  $t = e_1 e_2 \dots e_n$ 
 $next[10] \leftarrow (r \in e_n);$ 
 $next[9] \leftarrow next[10];$ 
 $next[8] \leftarrow (q \in e_n);$ 
 $next[7] \leftarrow next[8] \text{ implies } next[9];$ 
 $next[6] \leftarrow (q \in e_n);$ 
 $next[5] \leftarrow (p \in e_n);$ 
 $next[4] \leftarrow next[7];$ 
 $next[3] \leftarrow next[6];$ 
 $next[2] \leftarrow next[3] \text{ implies } next[4];$ 
 $next[1] \leftarrow next[2];$ 
for  $i = n - 1$  downto 1 do {
     $now[10] \leftarrow (r \in e_i);$ 
     $now[9] \leftarrow next[10];$ 
     $now[8] \leftarrow (q \in e_i);$ 
     $now[7] \leftarrow now[8] \text{ implies } now[9];$ 
     $now[6] \leftarrow (q \in e_i);$ 
     $now[5] \leftarrow (p \in e_i);$ 

```

```

    now[4] ← now[7] or next[4];
    now[3] ← now[6] or (now[5] and next[3]);
    now[2] ← now[3] implies now[4];
    now[1] ← now[2] and next[1];
    next ← now }
output(next[1]);

```

The algorithm above can be further optimized, noticing that only the bits 10, 4, 3 and 1 are needed in the vectors *now* and *next*, as we did for past time LTL in [130, 129]. The analysis of this algorithm is straightforward. Its time complexity is $\Theta(n \cdot m)$ while the memory required is $2 \cdot m$ bits, where n is the length of the trace and m is the size of the LTL formula.

8.2.2 Generating Dynamic Programming Algorithms

We now formally describe our algorithm that synthesizes dynamic programming algorithms like the one above from LTL formulae. Our synthesizer is generic, the potential user being expected to adapt it to his/her desired target language. The algorithm consists of three main steps:

Breadth First Search. The LTL formula should be first visited in breadth-first search (BFS) order to assign increasing numbers to subformulae as they are visited. Let $\varphi_1, \varphi_2, \dots, \varphi_m$ be the list of all subformulae in BFS order. Because of the semantics of finite trace LTL, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i+1} \models \varphi_{j'}$ for all $j \leq j' \leq m$. This recurrence gives the order in which one should generate the code.

Loop Initialization. Before we generate the “for” loop, we should first initialize the vector *next*[1..*m*], which basically gives the truth values of the subformulae on the empty trace. According to the semantics of LTL, one should fill the vector *next* backwards. For a given $m \geq j \geq 1$, *next*[*j*] is calculated as follows:

- If φ_j is a variable then $\text{next}[j] = (\varphi_j \in e_n)$. In a more complex setting, where φ_j was a state predicate, one would have to evaluate φ_j in the final state in the execution trace;
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then $\text{next}[j] = \text{not } \text{next}[j']$, where *not* is the negation operation on Booleans (bits);

- If φ_j is $\varphi_{j_1} \text{ Op } \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{next}[j] = \text{next}[j_1] \text{ op } \text{next}[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$, $\Box\varphi_{j'}$, or $\langle\rangle\varphi_{j'}$, then clearly $\text{next}[j] = \text{next}[j']$ according to the stationary semantics of our finite trace LTL;
- If φ_j is $\varphi_{j_1} \cup \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{next}[j] = \text{next}[j_2]$ for the same reason as above.

Loop Generation. Because of the dependences in the recursive definition of finite trace LTL satisfaction relation, one is expected to visit the remaining of the trace backwards, so the loop index will vary from $n - 1$ down to 1. The loop body will update/calculate the vector *now* and in the end will move it into the vector *next* to serve as basis for the next iteration. At a certain iteration i , the vector *now* is updated also backwards as follows:

- If φ_j is a variable then $\text{now}[j] = (\varphi_j \in e_i)$.
- If φ_j is $!\varphi_{j'}$ for some $j < j' \leq m$, then $\text{now}[j] = \text{not } \text{now}[j']$;
- If φ_j is $\varphi_{j_1} \text{ Op } \varphi_{j_2}$ for $j < j_1, j_2 \leq m$, then $\text{now}[j] = \text{now}[j_1] \text{ op } \text{now}[j_2]$, where Op is any propositional operation and op is its corresponding Boolean operation;
- If φ_j is $\circ\varphi_{j'}$ then $\text{now}[j] = \text{next}[j']$ since φ_j holds now if and only if $\varphi_{j'}$ held at the previous step (which processed the next event, the $i + 1$ -th);
- If φ_j is $\Box\varphi_{j'}$ then $\text{now}[j] = \text{now}[j']$ and $\text{next}[j]$ because φ_j holds now if and only if $\varphi_{j'}$ holds now and φ_j held at the previous iteration;
- If φ_j is $\langle\rangle\varphi_{j'}$ then $\text{now}[j] = \text{now}[j']$ or $\text{next}[j]$ because of similar reasons as above;
- If φ_j is $\varphi_{j_1} \cup \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\text{now}[j] = \text{now}[j_2]$ or $(\text{now}[j_1] \text{ and } \text{next}[j_1])$.

After each iteration, $\text{next}[1]$ says whether the initial LTL formula is validated by the trace $e_i e_{i+1} \dots e_n$. Therefore, the desired output is $\text{next}[1]$ after the last iteration. Putting all the above together, one can now write up the generic pseudocode presented below which can be implemented very efficiently on any current platform. Since the BFS procedure is linear,

the algorithm synthesizes a dynamic programming algorithm from an LTL formula in linear time and of linear size with the size of the formula.

The following generic program implements the discussed technique. It takes as input an LTL formula and generates a “for” loop which traverses the trace of events backwards, thus validating or invalidating the formula.

```

INPUT: LTL formula  $\varphi$ 
output("INPUT: trace  $t = e_1e_2...e_n$ ");
let  $\varphi_1, \varphi_2, \dots, \varphi_m$  be all the subformulae of  $\varphi$  in BFS order
for  $j = m$  downto 1 do {
    output("next",  $j$ , "["  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output(( $\varphi_j, " \in e_n$ "););
    if  $\varphi_j = !\varphi_{j'}$  then output("not next",  $j'$ , "[";););
    if  $\varphi_j = \varphi_{j_1} \text{ Op } \varphi_{j_2}$  then output("next",  $j_1$ , "[" op next",  $j_2$ , "[";););
    if  $\varphi_j = \circ\varphi_{j'}$  then output("next",  $j'$ , "[";););
    if  $\varphi_j = \Box\varphi_{j'}$  then output("next",  $j'$ , "[";););
    if  $\varphi_j = \langle\rangle\varphi_{j'}$  then output("next",  $j'$ , "[";););
    if  $\varphi_j = \varphi_{j_1} \cup \varphi_{j_2}$  then output("next",  $j_2$ , "[";);); }
output("for  $i = n - 1$  downto 1 do {");
for  $j = m$  downto 1 do {
    output("now",  $j$ , "["  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output(( $\varphi_j, " \in e_i$ "););
    if  $\varphi_j = !\varphi_{j'}$  then output("not now",  $j'$ , "[";););
    if  $\varphi_j = \varphi_{j_1} \text{ Op } \varphi_{j_2}$  then output("now",  $j_1$ , "[" op now",  $j_2$ , "[";););
    if  $\varphi_j = \circ\varphi_{j'}$  then output("next",  $j'$ , "[";););
    if  $\varphi_j = \Box\varphi_{j'}$  then output("now",  $j'$ , "[" and next",  $j$ , "[";););
    if  $\varphi_j = \langle\rangle\varphi_{j'}$  then output("now",  $j'$ , "[" or next",  $j$ , "[";););
    if  $\varphi_j = \varphi_{j_1} \cup \varphi_{j_2}$  then output("now",  $j_2$ , "[" or (now",  $j_1$ , "[" and
next",  $j$ , "[";);); }
output("next  $\leftarrow$  now; }");
output("output next[1];");

```

where *Op* is any propositional connective and *op* is its corresponding Boolean operator.

The Boolean operations used above are usually very efficiently implemented on any microprocessor and the vectors of bits *next* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions

in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast.

The dynamic programming technique presented in this section is as efficient as one can hope, but, unfortunately, has a major drawback: it needs to traverse the execution trace backwards. From a practical perspective, that means that the instrumented program is run for some period of time while its execution trace is saved, and then, after the program was stopped, its execution trace is traversed backwards and (efficiently) analyzed. Besides the obvious inconvenience due to storing potentially huge execution traces, this method cannot be used to monitor programs synchronously.

8.3 A Forwards and Often Synchronous Algorithm

In this section we shall present a more efficient rewriting semantics for LTL, based on the idea of consuming the events in the trace, one by one, and updating a data structure (which is also a formula) corresponding to the effect of the event on the value of the formula. An important advantage of this algorithm is that it often detects when a formula is violated or validated before the end of the execution trace, so, unlike the algorithms above, it is suitable for online monitoring. Our decision to write an operational semantics this way was motivated by an attempt to program such an algorithm in Java, where such a solution would be natural. The presented rewriting-based algorithm is linear in the size of the execution trace and worst-case exponential in the size of the monitored LTL formula.

8.3.1 An Event Consuming Algorithm

We will implement this rewriting based algorithm by extending the definition of the event consuming operation $_ \{ _ \} : \text{Formula Event}^* \rightarrow \text{Formula}$ to temporal operators, with the following intuition. Assuming a trace E, T consisting of event E followed by trace T , a formula X holds on this trace if and only if $X\{E\}$ holds on the remaining trace T . If the event E is terminal then $X\{E \ * \}$ holds if and only if X holds under standard LTL semantics on the infinite trace containing only the event E .

```
fmod LTL-REVISED is
  protecting LTL .
  vars X Y : Formula .
```

```

var E : Event .
var T : Trace .
eq (o X){E} = X .
eq (o X){E *} = X{E *} .
eq (<> X){E} = X{E} \/ <> X .
eq (<> X){E *} = X{E *} .
eq ([] X){E} = X{E} /\ [] X .
eq ([] X){E *} = X{E *} .
eq (X U Y){E} = Y{E} \/ X{E} /\ X U Y .
eq (X U Y){E *} = Y{E *} .

op _|-_ : Trace Formula -> Bool .
eq E |- X = [X{E *}] .
eq E,T |- X = T |- X{E} .
endfm

```

The rule for the temporal operator $\Box X$ should be read as follows: the formula X must hold now ($X\{E\}$) and also in the future ($\Box X$). The sub-expression $X\{E\}$ represents the formula that must hold on the rest of the trace in order for X to hold now.

As an example, consider again the traffic light controller safety formula $\Box(\text{green} \rightarrow \text{!red} \cup \text{yellow})$, which is first rewritten to $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$ by the equations in module PROP-CALC. This formula modified by an event `green yellow` (notice that two lights can be lit at the same time) yields the rewriting sequence

```

( $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})\{\text{green yellow}\}$  ==>
( $\text{true} ++ \text{green}\{\text{green yellow}\}$ 
  ++  $\text{green}\{\text{green yellow}\} /\ ((\text{true} ++ \text{red}) \cup \text{yellow})\{\text{green yellow}\}$ 
  /\  $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$  ==>
( $(\text{true} ++ \text{red}) \cup \text{yellow}\}\{\text{green yellow}\}$ 
  /\  $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$  ==>
( $\text{yellow}\{\text{green yellow}\} \cup ((\text{true} ++ \text{red}\{\text{green yellow}\}) /\ (\text{true} ++ \text{red}) \cup \text{yellow})$ 
  /\  $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$  ==>
 $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$ 

```

which is exactly the original formula, while the same formula transformed by just the event `green` yields

```

( $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})\{\text{green}\}$  ==>
( $\text{true} ++ \text{green}\{\text{green}\}$ 
  ++  $\text{green}\{\text{green}\} /\ ((\text{true} ++ \text{red}) \cup \text{yellow})\{\text{green}\}$ 
  /\  $\Box(\text{true} ++ \text{green} ++ \text{green} /\ (\text{true} ++ \text{red}) \cup \text{yellow})$  ==>
( $(\text{true} ++ \text{red}) \cup \text{yellow}\}\{\text{green}\}$ 

```

```

      /\ [](true ++ green ++ green /\ (true ++ red) U yellow)      ==>
(yellow{green} \/ ((true ++ red{green}) /\ (true ++ red) U yellow)
      /\ [](true ++ green ++ green /\ (true ++ red) U yellow)      ==>
(true ++ red) U yellow /\ [](true ++ green ++ green /\ (true ++ red) U yellow)

```

which further modified by an event red yields

```

(yellow{red} \/ (true ++ red{red}) /\ (true ++ red) U yellow)
  /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red}  ==>
false /\ ([](true ++ green ++ green /\ (true ++ red) U yellow)){red} ==>
false

```

When the current formula becomes **false**, as it happened above, we say that the original formula has been violated. Indeed, the current formula will remain **false** for any subsequent trace of events, so the result of the monitoring session will be **false**.

Note that the rewriting system described so far obviously terminates, because what it does is to propagate the current event to the atomic sub-formulae, replace those by either true or false, and eventually canonize the newly obtained formula.

Some operators could be defined in terms of others, as is typically the case in the standard semantics for LTL. For example, we could introduce an equation of the form: $\langle \rangle X = \text{true} \cup X$, and then eliminate the rewriting rule for $\langle \rangle X$ in the above module. This turns out to be less efficient in practice though, because more rewrites are needed. This happens regardless of whether one enables memoization (explained in detail in Subsection 8.3.3) or not, because memoization brings a real benefit only when previously processed terms are to be reduced again.

This module eventually defines a new satisfaction relation $_|-_$ between traces and formulae. The term $T \mid - X$ is evaluated now by an iterative traversal over the trace, where each event transforms the formula. Note that the new formula that is generated at each step is always kept small by being reduced to normal form via the equations in the PROP-CALC module in Subsection 2.1.2.

Our current JPAX implementation of the rewriting algorithm above executes the last two rules of the module LTL-REVISED outside the main rewriting engine. More precisely, Maude is started in its *loop mode* [76], which provides the capability of enabling rewriting rules in a reactive system style: a “state” term is stored, which can then be modified via rewriting rules that are activated by ASCII text events that are provided via the standard I/O. JPAX starts a Maude process and assigns the formula to be monitored

as its loop mode state. Then, as events are received from the monitored program, they are filtered and forwarded to the Maude module, which then enables rewriting on the term $X\{E\}$, where X is the current formula and E is the newly received event; the normal form of this reduction, a formula, is stored as the new loop mode state term. The process continues until the last event is received. JPAX tags the last event, asking Maude to reduce a term $X\{E \text{ *}\}$; the result will be either **true** or **false**, which is reported to the user at the end of the monitoring session².

A natural question here is how big the stored formula can grow during a monitoring session. Such a formula will consist of Boolean combinations of sub-formulae of the initial formula, kept in a minimal canonical form. This can grow exponentially in the size of the initial formula in the worst-case (see [219] for a related result for extended regular expressions).

Theorem 15 *For any formula X of size m and any sequence of events to be monitored E_1, E_2, \dots, E_n , the formula $X\{E_1\}\{E_2\} \dots \{E_n\}$ needs $O(2^m)$ space to be stored. Moreover, the exponential space cannot be avoided: any synchronous or asynchronous forwards monitoring algorithm for LTL requires space $\Omega(2^{c\sqrt{m}})$ space, where c is some fixed constant.*

Proof: Due to the Boolean ring simplification rules in PROP-CALC, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions, where conjuncts have temporal operators at top. Moreover, after processing any number of events, in our case E_1, E_2, \dots, E_n , the conjuncts in the normal form of $X\{E_1\}\{E_2\} \dots \{E_n\}$ are subterms of the initial formula X , each having a temporal operator at its top. Since there are at most m such subformulae of X , it follows that there are at most 2^m possibilities to combine them in a conjunction. Therefore, one needs space $O(2^m)$ to store any exclusive disjunction of such conjunctions. This reasoning only applies on “idealistic” rewriting engines, which carefully optimize space needs during rewriting. It is not clear to us whether Maude is able to attain this space upper bound in all situations.

For the space lower bound of any finite trace LTL monitoring algorithm, consider a simplified framework with only two atomic predicate and therefore only four possible states. For simplicity, we encode these four states by 0, 1, # and \$. Consider also some natural number k and the language

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}.$$

²In fact, JPAX reports a similar message also when the current monitoring requirement becomes **true** or **false** at any time during the monitoring process.

This language was previously used in several works [171, 172, 219] to prove lower bounds. The language can be shown to contain exactly those finite traces satisfying the following LTL formula [172] of size $\Theta(k^2)$:

$$\phi_k = [(!\$)\mathcal{U}(\$ \wedge \circ\Box(!\$)) \wedge \Diamond[\# \wedge \circ^{n+1}\# \wedge \bigwedge_{i=1}^n ((\circ^i 0 \wedge \Box(\$ \rightarrow \circ^i 0)) \vee (\circ^i 1 \wedge \Box(\$ \rightarrow \circ^i 1)))].$$

Let us define an equivalence relation on finite traces in $(0 + 1 + \#)^*$. For a $\sigma \in (0 + 1 + \#)^*$, define $S(\sigma) = \{w \in (0 + 1)^k \mid \exists \lambda_1, \lambda_2. \lambda_1 \# w \# \lambda_2 = \sigma\}$. We say that $\sigma_1 \equiv_k \sigma_2$ if and only if $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of \equiv_k is 2^{2^k} ; this is because for any $S \subseteq (0 + 1)^k$, there is a σ such that $S(\sigma) = S$.

Since $|\phi_k| = \Theta(k^2)$, it follows that there is some constant c' such that $|\phi_k| \leq c'k^2$ for all large enough k . Let c be the constant $1/\sqrt{c'}$. We will prove this lower bound result by contradiction. Suppose \mathcal{A} is an LTL forwards monitoring algorithm that uses less than $2^{c\sqrt{m}}$ space for any LTL formulae of large enough size m . We will look at the behavior of the algorithm \mathcal{A} on inputs of the form ϕ_k . So $m = |\phi_k| \leq c'k^2$, and \mathcal{A} uses less than 2^k space. Since the number of equivalence classes of \equiv_k is 2^{2^k} , by the pigeon hole principle there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the memory of \mathcal{A} on ϕ_k after reading $\sigma_1 \$$ is the same as the memory after reading $\sigma_2 \$$. In other words, \mathcal{A} running on ϕ_k will give the same answer on all traces of the form $\sigma_1 \$w$ and $\sigma_2 \$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2)) \cup (S(\sigma_2) \setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1 \$w$ and $\sigma_2 \$w$ is in L_k , and so \mathcal{A} running on ϕ_k gives the wrong answer on one of these inputs. Therefore, \mathcal{A} is not a correct. \square

It seems, however, that this worst-case exponential complexity in the size of the LTL formula is more of theoretical importance than practical, since in general the size of the formula rarely grew more than twice in our experiments. Verification results are very encouraging and show that this optimized semantics is orders of magnitude faster than the first semantics. Traces of less than 10,000 events are verified in milliseconds, while traces of 100,000 events never needed more than 3 seconds. This technique scales quite well; we were able to monitor even traces of hundreds of millions events. As a concrete example, we created an artificial trace by repeating 10 million times the 10 event trace in Subsection 8.1.2, and then checked it against the formula $\Box(\text{green} \rightarrow !\text{red} \cup \text{yellow})$. There were needed 4.9 billion rewriting steps for a total of about 1,500 seconds. In Subsection 8.3.3 we will see how this algorithm can be made even more efficient, using memoization.

8.3.2 Correctness and Completeness

In this subsection we prove that the algorithm presented above is correct and complete with respect to the semantics of finite trace LTL presented in Section 8.1. The proof is done completely in Maude, but since Maude is not intended to be a theorem prover, we actually have to generate the proof obligations by hand. In other words, the proof that follows was *not* generated automatically. However, it could have been mechanized by using proof assistants and/or theorem provers like KUMO [115], PVS [245], or Maude-ITP [69]. We have already done it in PVS, but we prefer to use only plain Maude in this paper.

Theorem 16 *For any trace T and any formula X , $T \models X$ if and only if $T \vdash X$.*

Proof: By induction, both on traces and formulae. We first need to prove two lemmas, namely that the following two equations hold in the context of both LTL and LTL-REVISED:

$$\begin{aligned} (\forall E : \text{Event}, X : \text{Formula}) \quad E \models X &= E \vdash X, \\ (\forall E : \text{Event}, T : \text{Trace}, X : \text{Formula}) \quad E, T \models X &= T \models X\{E\}. \end{aligned}$$

We prove them by structural induction on the formula X . Constants e and x are needed in order to prove the first lemma via the theorem of constants. However, since we prove these lemmas by structural induction on X , we not only have to add two constants e and t for the universally quantified variables E and T , but also two other constants y and z standing for formulas which can be combined via operators to give other formulas. The induction hypotheses are added to the following specification via equations. Notice that we merged the two proofs to save space. A proof assistant like KUMO, PVS or Maude-ITP would prove them independently, generating only the needed constants for each of them.

```
fmod PROOF-OF-LEMMAS is
  extending LTL .
  extending LTL-REVISED .
  op e : -> Event .
  op t : -> Trace .
  ops a b c : -> Atom .
  ops y z : -> Formula .
  eq e \models y = e \vdash y .
  eq e \models z = e \vdash z .
```

```

eq e,t |= y = t |= y{e} .
eq e,t |= z = t |= z{e} .
eq b{e} = true .
eq c{e} = false .
endfm

```

It is worth reminding the reader at this stage that the functional modules in Maude have initial semantics, so proofs by induction are valid. Before proceeding further, the reader should be aware of the operational semantics of the operation `_==_`, namely that the two argument terms are first reduced to their normal forms which are then compared syntactically (but modulo associativity and commutativity); it returns `true` if and only if the two normal forms are equal. Therefore, the answer `true` means that the two terms are indeed semantically equal, while `false` only means that they could not be proved equal; they can still be equal.

```

reduce (e |= a      == e |- a)
  and (e |= true    == e |- true)
  and (e |= false   == e |- false)
  and (e |= y /\ z == e |- y /\ z)
  and (e |= y ++ z == e |- y ++ z)
  and (e |= [] y    == e |- [] y)
  and (e |= <> y     == e |- <> y)
  and (e |= y U z   == e |- y U z)
  and (e |= o y     == e |- o y)

  and (e,t |= true  == t |= true{e})
  and (e,t |= false == t |= false{e})
  and (e,t |= b     == t |= b{e})
  and (e,t |= c     == t |= c{e})
  and (e,t |= y /\ z == t |= (y /\ z){e})
  and (e,t |= y ++ z == t |= (y ++ z){e})
  and (e,t |= [] y   == t |= ([] y){e})
  and (e,t |= <> y   == t |= (<> y){e})
  and (e,t |= y U z  == t |= (y U z){e})
  and (e,t |= o y    == t |= (o y){e}) .

```

It took Maude 129 reductions to prove these lemmas. Therefore, one can safely add now these lemmas as follows:

```

fmod LEMMAS is
  protecting LTL .
  protecting LTL-REVISED .
  var E : Event .

```

```

var T : Trace .
var X : Formula .
eq E |= X = E |- X .
eq E,T |= X = T |- X{E} .
endfm

```

We can now prove the theorem, by induction on traces. More precisely, we show:

$\mathcal{P}(E)$, and
 $\mathcal{P}(T)$ implies $\mathcal{P}(E,T)$, for all events E and traces T ,

where $\mathcal{P}(T)$ is the predicate “for all formulas X , $T \models X$ iff $T \vdash X$ ”. This induction schema can be easily formalized in Maude as follows:

```

fmod PROOF-OF-THEOREM is
  protecting LEMMAS .
  op e : -> Event .
  op t : -> Trace .
  op x : -> Formula .
  var X : Formula .
  eq t |= X = t |- X .
endfm

reduce e |= x == e |- x .
reduce e,t |= x == e,t |- x .

```

Notice the difference in role between the constant x and the variable X . The first reduction proves the base case of the induction, using the theorem of constants for the universally quantified variable X . In order to prove the induction step, we first applied the theorem of constants for the universally quantified variables E and T , then added $\mathcal{P}(t)$ to the hypothesis (the equation “ $\text{eq } t \models X = t \vdash X$.”), and then reduced $\mathcal{P}(e \ t)$ using again the theorem of constants for the universally quantified variable X . Like in the proofs of the lemmas, we merged the two proofs to save space. \square

8.3.3 Further Optimization by Memoization

Even though the formula transforming algorithm in Subsection 8.3.1 can process 100 million events in about 25 minutes, which is relatively reasonable for practical purposes, it can be significantly improved by adding only 5 more characters to the existing Maude code presented so far. More precisely, one can replace the operation declaration

```
op _{} : Formula Event* -> Formula [prec 10]
```

in module LOGICS-BASIC by the operation declaration

```
op _{} : Formula Event* -> Formula [memo prec 10]
```

The attribute `memo` added to an operation declaration instructs Maude to memorize, or cache, the normal forms of terms rooted in that operation, i.e., those terms will be rewritten only once. Memoization is implemented by hashing, where the entry in the hash table is given by the term to be reduced and the value in the hash is its normal form. In our concrete example, memoization has the effect that any LTL formula will be transformed by a given event exactly once during the monitoring sequence; if the same formula and the same event occur in the future, the resulting modified formula is extracted from the hash table without applying any rewriting step. If one thinks of LTL in terms of automata, then our new algorithm corresponds to building the monitoring automaton *on the fly*. The obvious benefit of this technique is that only the *needed* part of the automaton is built, namely that part that is reachable during monitoring a particular sequence of events, which is practically very useful because the entire automaton associated to an LTL formula can be exponential in size, so storing it might become a problem.

The use of memoization brings a significant improvement in the case of LTL. For example, the same sequence of 100 million events, which took 1500 seconds using the algorithm presented in Subsection 8.3.1, takes only 185 seconds when one uses memoization, for a total of 2.2 rewritings per processed event and 540,000 events processed per second! We find these numbers amazingly good for any practical purpose we can think of and believe that, taking into account the simplicity, obvious correctness and elegance of the rewriting based algorithm (implemented basically by 8 rewriting rules in LTL-REVISED), it would be hard to argue for any other implementation of LTL monitoring. One should, however, be careful when one uses memoization because hashing slows down the rewriting engine. LTL is a happy case where memoization brings a significant improvement, because the operational semantics of all the operators can be defined recursively, so formulae repeat often during the monitoring process. However, there might be monitoring logics where memoization could be less efficient. Such a logic would probably be an extension of LTL with time, allowing formulae of the form “ $\langle 5 \rangle X$ ” with the meaning “eventually in 5 units of time X ”, because of a potentially very large number of terms to be memoized: $\langle 5 \rangle X$, $\langle 4 \rangle X$, etc. Experimentation

is certainly needed if one designs a new logic for monitoring and wants to use memoization.

8.4 Generating Forwards, Synchronous and Efficient Monitors

Even though the rewriting based monitoring algorithm presented in the previous section performs quite well in practice, there can be situations in which one wants to minimize the monitoring overhead as much as possible. Additionally, despite its simplicity and elegance, the procedure above requires an efficient rewriting engine modulo associativity and commutativity, which may not be available or may not be desirable on some monitoring platforms, such as, for example, within an embedded system.

In this section we give a technique, based on ideas presented previously in the paper, to generate automata-based optimal monitors for future time LTL formulae. By optimality we here mean everything one may expect, such as minimal number of states, forwards traversal of execution traces, synchronicity, efficiency, but also less standard optimality features, such as transiting from one state to another with a minimum amount of computation. In order to effectively do this we introduce the notion of *binary transition tree* (BTT), as a generalization of binary decision diagrams (BDD) [46], whose purpose is to provide an *optimal order* in which state predicates need to be evaluated to decide the next state. The motivation for this is that in practical applications evaluating a state predicate is a time consuming task, such as for example to check whether a vector is sorted. The associated finite state machines are called *binary transition tree finite state machines* (BTT-FSM).

The drawback of generating an optimal BTT-FSM statically, i.e., before monitoring, is the worst-case double exponential time/space required at startup. Therefore, the algorithm presented in this section is recommended for situations where the LTL formulae to monitor are relatively small in size but the runtime overhead is desired to be minimal. It is worth noting that the BTT-FSM generation process can potentially take place on a machine different from the one performing the monitoring. In particular, one can think of a WWW fast server offering LTL-to-BTT-FSM services via the Internet, which can also maintain a database of already generated BTT-FSMs to avoid regenerating the same monitors.

```

1. let  $S$  be  $\varphi$ 
2. procedure LTL2MT-FSM( $\varphi$ )
3.   let  $\mu^*(\varphi)$  be  $\emptyset$ 
4.   let  $\mu(\varphi)$  be  $\emptyset$ 
5.   foreach  $\theta : A \rightarrow \{true, false\}$  do
6.     let  $e_\theta$  be the list of atoms  $a$  with  $\theta(a) = true$ 
7.     let  $p_\theta$  be the proposition  $\bigwedge\{a \mid \theta(a) = true\} \wedge \bigwedge\{\neg a \mid \theta(a) = false\}$ 
8.     let  $\mu^*(\varphi)$  be MERGE( $[p_\theta ? \varphi\{e_\theta^*\}], \mu^*(\varphi)$ )
9.     let  $\varphi_\theta$  be  $\varphi\{e_\theta\}$ 
10.    if there is  $\varphi' \in S$  with  $\text{VALID}(\varphi_\theta \leftrightarrow \varphi')$ 
11.      then let  $\mu(\varphi)$  be MERGE( $[p_\theta ? \varphi']$ ,  $\mu(\varphi)$ )
12.    else let  $S$  be  $S \cup \{\varphi_\theta\}$ 
13.      let  $\mu(\varphi)$  be MERGE( $[p_\theta ? \varphi_\theta]$ ,  $\mu(\varphi)$ )
14.      LTL2MT-FSM( $\varphi_\theta$ )
15.    endfor
16.    if  $\mu(\varphi) = [true ? \varphi]$  and  $\mu^*(\varphi) = [true ? b]$  then replace  $\varphi$  by  $b$  everywhere
17. endprocedure

```

Figure 8.1: Algorithm to generate a minimal MT-FSM* $(S, A, \mu, \mu^*, \varphi)$ from an LTL formula φ .

8.4.1 From LTL Formulae to BTT-FSMs

Informally, our algorithm to generate optimal BTT-FSMs from LTL formulae consists of two steps. First, it generates an MT-FSM with a minimum number of states. Then, using the technique presented in the previous subsection, it generates an optimal BTT from each MT.

To generate a minimal MT-FSM, our algorithm uses the rewriting based procedure presented in Section 8.3 on all possible events, until the set of formulae to which the original LTL formula can “evolve” stabilizes. The procedure LTL2MT-FSM shown in Figure 8.1 builds an MT-FSM whose states are formulae, with the help of a validity checker. Initially, the set S of states contains only the original LTL formula. LTL2MT-FSM is called on the original LTL formula, and then recursively in a depth-first manner on all the formulae to which the initial formula can ever evolve via the event-consuming operator $[_{-}]$ introduced in Section 8.3.

For each LTL state formula φ in S , multi-transitions $\mu(\varphi)$ and $\mu^*(\varphi)$ are maintained. For each possible event θ , one first updates $\mu^*(\varphi)$ by considering the case in which θ is the last event in a trace (step 8), and then the current formula φ evolves into the corresponding formula φ_θ (step 9). If some equivalent formula φ' to φ_θ has already been discovered then one only needs to modify the multi-transition set of φ accordingly in order to point to φ' (step 11). Notice that equivalence of LTL formulae is checked by using a validity procedure (step 10), which is given in Figure 8.2. If there is no formula in S equivalent to φ_θ , then the new formula φ_θ is added to S , multi-transition $\mu(\varphi)$ is updated accordingly, and then the MT-FSM generation procedure is called recursively on φ_θ . This way, one eventually generates all possible LTL formulae into which the initial formula can ever evolve during a monitoring session; this happens, of course, modulo finite trace LTL semantic equivalence, implemented elegantly using the validity checker described below. By Theorem 15, this recursion will eventually terminate, leading to an MT-FSM*.

The procedure `VALID` used in LTL2MT-FSM above is defined in Figure 8.2. It essentially follows the same idea of generating all possible formulae to which the original LTL formula tested for validity can evolve via the event consumption operator defined by rewriting in Section 8.3, but for each newly obtained formula φ and for each event θ , it also checks whether an execution trace stopping at that event would be a rejecting sequence. The intuition for this check is that a formula is valid under finite trace LTL semantics if and only if any (finite) sequence of events satisfies that formula; since any generated formula corresponds to one into which the initial formula can evolve, we need to make sure that each of these formulae becomes *true* under any possible last monitored event. This is done by rewriting the term $\varphi\{e_\theta^*\}$ with the rules in Section 8.3 to its normal form (step 5.), i.e., *true* or *false*. The formula is valid if and only if there is no rejecting sequence; the entire space of evolving formulae is again explored by depth-first search. Notice that `VALID` does not test for equivalence of formulae, so it can potentially generate a larger number of formulae than LTL2MT-FSM. However, by Theorem 15, this procedure will also eventually terminate.

Theorem 17 *Given an LTL formula φ of size m , the following hold:*

1. *The procedure `VALID` is correct, that is, `VALID`(φ) returns *true* if and only if φ is satisfied, as defined in Section 8.1, by any finite trace;*
2. *The space and time complexity of `VALID`(φ) is $2^{O(2^m)}$;*

```

1. let  $S$  be  $\varphi$ 
2. function VALID( $\varphi$ )
3.   foreach  $\theta : A \rightarrow \{true, false\}$  do
4.     let  $e_\theta$  be the list of atoms  $a$  with  $\theta(a) = true$ 
5.     if  $\varphi\{e_\theta^*\} = false$  then return false
6.     let  $\varphi_\theta$  be  $\varphi\{e_\theta\}$ 
7.     if  $\varphi_\theta \notin S$ 
8.       then let  $S$  be  $S \cup \{\varphi_\theta\}$ 
9.       if VALID( $\varphi_\theta$ ) = false then return false
10.   endfor
11.   return true
12. end function

```

Figure 8.2: Validity checker for an LTL formula φ .

Additionally, letting M denote the MT-FSM* $(S, A, \mu, \mu^*, \varphi)$ generated by LTL2MT-FSM(φ), the following hold:

3. LTL2MT-FSM(φ) is correct; more precisely, M has the property that for any events $\theta_1, \dots, \theta_n, \theta$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta^*} true$ if and only if the finite trace $e_{\theta_1} \dots e_{\theta_n} e_\theta$ satisfies φ , and $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta^*} false$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ does not satisfy φ ;
4. M is synchronous; more precisely, for any events $\theta_1, \dots, \theta_n$, it is the case that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta} true$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a valid prefix of φ , and $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{e_\theta} false$ if and only if $e_{\theta_1} \dots e_{\theta_n} e_\theta$ is a bad prefix of φ ;
5. The space and time complexity of LTL2MT-FSM(φ) is $2^{O(2^m)}$;
6. M is the smallest MT-FSM* which is correct and synchronous (as defined in 3 and 4 above);
7. Monitoring against a BTT-FSM* corresponding to M , as shown in Subsection 6.1.2, needs $O(2^m)$ space and time.

Proof: 1. Using a depth-first search strategy, the procedure VALID visits all possible formulae into which the original formula φ can evolve via

any possible sequence of events. These formulae are different modulo the rewriting rules in Section 8.3 which are used to simplify LTL formulae and to remove the derivatives, but those rules were shown to be sound, so VALID indeed explores all possible LTL formulae into which φ can *semantically* evolve. Moreover, for each such formula, say φ' , and each event, VALID checks at Step 5 whether a trace terminating with that event in φ' would lead to rejection. VALID(φ) returns true if and only if there is no such rejection, so it is indeed correct: if it had been some finite trace that did not satisfy φ then VALID would have found it during its exhaustive search.

2. The space required by VALID(φ) is clearly dominated by the size of S . By Theorem 15, each formula φ' into which φ can evolve needs $O(2^m)$ space to be stored. That means that there can be at most $2^{O(2^m)}$ such formulae. So the total space needed to store S in the worst case is $2^{O(2^m)}$. An amortized analysis of its running time, tells us that VALID runs its “for each event” loop one per formula in S . Since the number of events is much less than 2^m and since reducing the derivative of a formula by an event takes time proportional with the size of the formula, we deduce that the total running time of the VALID procedure is $2^{O(2^m)}$.

3. By Theorem 16 and the event consuming procedure described in Subsection 8.3.1, it follows that $e_{\theta_1} \dots e_{\theta_n} e_{\theta}$ satisfies φ if and only if $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_{\theta}^*\}$ reduces to *true*, and that $e_{\theta_1} \dots e_{\theta_n} e_{\theta}$ does not satisfy φ if and only if $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_{\theta}^*\}$ reduces to *false*. It is easy to see that VALID($\psi \leftrightarrow \psi'$) returns true if and only if ψ and ψ' are formulae equivalent under the finite trace LTL semantics. That means the MT-FSM* generated by LTL2MT-FSM(φ) contains a formula-state φ_1 which is equivalent to $\varphi\{e_{\theta_1}\}$; moreover, $\varphi \xrightarrow{\theta_1} \varphi_1$ in this MT-FSM*. Inductively, one can show that there is a series of formulae-states $\varphi_1, \dots, \varphi_n$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}$ is equivalent to φ_n . The rest follows by noting that $\varphi_n\{e_{\theta}^*\}$ reduces to true if and only if $\varphi_n \xrightarrow{\theta^*} \text{true}$ in the generated MT-FSM, and that $\varphi_n\{e_{\theta}^*\}$ reduces to false if and only if $\varphi_n \xrightarrow{\theta^*} \text{false}$ in the MT-FSM.

4. As in 3 above, one can inductively show that there is a series of formulae $\varphi_1, \dots, \varphi_n, \varphi'$ such that $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ is a sequence of transitions in the generated MT-FSM and $\varphi\{e_{\theta_1}\}\{\dots\}\{e_{\theta_n}\}\{e_{\theta}\}$ is equivalent to φ' . By Proposition 14, due to the use of the validity checker in step 10 of LTL2MT-

FSM it follows that $e_{\theta_1} \dots e_{\theta_n} e_{\theta}$ is a valid prefix of φ if and only if φ' is equivalent to *true*, and that $e_{\theta_1} \dots e_{\theta_n} e_{\theta}$ is a bad prefix of φ if and only if φ' is equivalent to *false*. The rest follows now by Step 16 in the algorithm in Figure 8.1, which, due to the validity checker, provides a necessary and sufficient condition for a processed formula to be semantically equivalent to *true* or *false*, respectively.

5. The space required by LTL2MT-FSM(φ) is again dominated by the size of S . Like in the analysis of VALID in §2 above, by Theorem 15 we get that there can be at most $2^{O(2^m)}$ formulae generated in S , so the total space needed to store S in the worst case is also $2^{O(2^m)}$. For each newly added formula in S and for each event, Step 10 calls the procedure VALID potentially once for each already existing formula in S . It is important to notice that the formulae on which VALID is called are exclusive disjunctions of conjunctions of sub-formulae rooted in temporal operators of the original formula φ , so its space and time complexity will also be $2^{O(2^m)}$ each time it is called by LTL2MT-FSM. One can easily see now that the space and time requirements of LTL2MT-FSM(φ) are also $2^{O(2^m)}$ (the constant in the exponent $O(2^m)$ can be appropriately enlarged).

6. For any MT-FSM* machine $M' = (S', A, \mu', \mu'^*, q_0)$, one can associate a formula, more precisely a state in M , to each state in S' as follows. φ is associated to the initial state q_0 . Then for each transition $q_1 \xrightarrow{\theta} q_2$ in M' such that q_1 has a formula φ_1 associated and q_2 has no formula associated, then one associates that φ_2 to q_2 with the property that $\varphi_1 \xrightarrow{\theta} \varphi_2$ is a transition in M . For a state q in S' let φ_q be the formula in S associated to it. Since M' is correct and synchronous for φ , it follows that $\mathcal{L}_{M'}(q) = \mathcal{L}_M(\varphi_q)$ for any state q in S' . We can now show that the map associating a formula in S to any state in S' is surjective, which shows that M has therefore a minimal number of states. Let φ' be a formula in S and let $\varphi \xrightarrow{\theta_1} \varphi_1 \dots \xrightarrow{\theta_n} \varphi_n \xrightarrow{\theta} \varphi'$ be a sequence of transitions in M leading to φ' ; note that all formulae in S are reachable via transitions in M from φ . Let q' be the state in S' such that $q_0 \xrightarrow{\theta_1} q_1 \dots \xrightarrow{\theta_n} q_n \xrightarrow{\theta} q'$. Then $\mathcal{L}_{M'}(q') = \mathcal{L}_M(\varphi_{q'})$. Since by Proposition 14, $\mathcal{L}_{M'}(q') = \{t \mid e_{\theta_1} \dots e_{\theta_n} e_{\theta} t \in \mathcal{L}_{M'}(q_0)\}$ and $\mathcal{L}_M(\varphi') = \{t \mid e_{\theta_1} \dots e_{\theta_n} e_{\theta} t \in \mathcal{L}_M(\varphi)\}$, it follows that $\mathcal{L}_M(\varphi_{q'}) = \mathcal{L}_M(\varphi')$, that is, that $\varphi_{q'}$ and φ' are equivalent. Since Step 10 of the LTL2MT-FSM eventually uses the validity checker on any pairs of formulae in S , it follows that $\varphi_{q'} = \varphi'$.

7. In order to distinguish N pieces of data, $\log(N)$ bits are needed to

encode each datum. Therefore, one needs $O(2^m)$ bits to encode a state of M , which is the main memory needed by the monitoring algorithm. Like in Theorem 15, we assume “idealistic” rewriting engines able to optimize the space requirements; we are not aware whether Maude is able to attain this performance or not. To make a transition from one state to another, a BTT-FSM* associated to the MT-FSM* generated by $\text{LTL2MT-FSM}(\varphi)$ needs to only evaluate at most all the atomic predicates occurring in the formula, which, assuming that evaluating atom predicates takes unit time, is clearly $O(2^m)$. When the entire BTT is evaluated, it finally has to store the newly obtained state of the BTT-FSM*, which can reuse the space of the previous one, $O(2^m)$, and which also takes time $O(2^m)$. \square

Once the procedure LTL2MT-FSM terminates, the formulae φ , φ' , etc., are not needed anymore, so one can and should replace them by unique labels in order to reduce the amount of storage needed to encode the MT-FSM* and/or the corresponding BTT-FSM*. This algorithm can be relatively easily implemented in any programming language. We have, however, found Maude again a very elegant system for this task, implementing the entire LTL formula to BTT-FSM algorithm in about 200 lines of code.

8.4.2 Examples

The BTT-FSM generation algorithm presented in this section, despite its overall worst-case high startup time, can be very useful when formulae are relatively short, as it is most often the case in practice. For the traffic light controller requirement formula discussed previously in the paper, $\Box(\text{green} \rightarrow (\neg \text{red}) \cup \text{yellow})$, this algorithm generates in about 0.2 seconds the optimal BTT-FSM* in Figure 8.3, also shown in Figure 6.3 in flowchart notation; Figure 6.2 shows its optimal MT-FSM*. For simplicity, the states *true* and

State	Non-terminal event	Terminal event
1	<code>yellow ? 1 : green ? red ? false : 2 : 1</code>	<code>yellow ? true : green ? false : true</code>
2	<code>yellow ? 1 : red ? false : 2</code>	<code>yellow ? true : false</code>

Figure 8.3: An optimal BTT-FSM* for the formula $\Box(\text{green} \rightarrow \neg \text{red} \cup \text{yellow})$.

false do not appear in Figure 8.3. Notice that the atomic predicate `red` does *not* need to be evaluated on terminal events and that `green` does not need

to be evaluated in state 2. In this example, the colors are not supposed to exclude each other, that is, the traffic controller can potentially be both green and red.

The LTL formulae on which our algorithm has the worst performance are those containing many nested temporal operators (which are not frequently used in specifications anyway, because of the high risk of getting them wrong). For example, it takes our Maude implementation of this algorithm 1.3 seconds to generate the minimal 3-state (*true* and *false* states are not counted) BTT-FSM* for the formula $a \cup (b \cup (c \cup d))$ and 13.2 seconds to generate the 7-state minimal BTT-FSM* for the formula $((a \cup b) \cup c) \cup d$. It never took our current implementation more than a few seconds to generate the BTT-FSM* of any LTL formula of interest for our applications, i.e., non-artificial. Figure 8.4 shows the generated BTT-FSM of some artificial LTL formulae, taking together less than 15 seconds to be generated. To keep the figure small, the states *true* and *false* together with their self-transitions are not shown in Figure 8.4, and they are replaced by **t** and **f** in BTTs.

Formula	State	Monitoring BTT	Terminating BTT
$\Box \Diamond a$	1	1	$a ? \mathbf{t} : \mathbf{f}$
$\Diamond(\Box a \vee \Box \neg a)$	—	—	—
$\Box(a \rightarrow \Diamond b)$	1 2	$a ? (b ? 1 : 2) : 1$ $b ? 1 : 2$	$a ? (b ? \mathbf{t} : \mathbf{f}) : \mathbf{t}$ $b ? \mathbf{t} : \mathbf{f}$
$a \mathcal{U} (b \mathcal{U} c)$	1 2	$c ? \mathbf{t} : (a ? 1 : (b ? 2 : \mathbf{f}))$ $c ? \mathbf{t} : (b ? 2 : \mathbf{f})$	$c ? \mathbf{t} : \mathbf{f}$ $c ? \mathbf{t} : \mathbf{f}$
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} d))$	1 2 3	$d ? \mathbf{t} : a ? 1 : b ? 2 : c ? 3 : \mathbf{f}$ $d ? \mathbf{t} : b ? 2 : c ? 3 : \mathbf{f}$ $d ? \mathbf{t} : c ? 3 : \mathbf{f}$	$d ? \mathbf{t} : \mathbf{f}$ $d ? \mathbf{t} : \mathbf{f}$ $d ? \mathbf{t} : \mathbf{f}$
$((a \mathcal{U} b) \mathcal{U} c) \mathcal{U} d$	1 2 3 4 5 6 7	$d ? \mathbf{t} : c ? 1 : b ? 4 : a ? 5 : \mathbf{f}$ $b ? c ? \mathbf{t} : 7 : a ? c ? 6 : 2 : \mathbf{f}$ $b ? d ? \mathbf{t} : c ? 1 : 4 : a ? d ? 6 : c ? 3 : 5 : \mathbf{f}$ $c ? d ? \mathbf{t} : 1 : b ? d ? 7 : 4 : a ? d ? 2 : 5 : \mathbf{f}$ $b ? d ? c ? \mathbf{t} : 7 : c ? 1 : 4 : a ? d ? c ? 6 : 2 : c ? 3 : 5 : \mathbf{f}$ $b ? \mathbf{t} : a ? 6 : \mathbf{f}$ $c ? \mathbf{t} : b ? 7 : a ? 2 : \mathbf{f}$	$d ? \mathbf{t} : \mathbf{f}$ $c ? b ? \mathbf{t} : \mathbf{f} : \mathbf{f}$ $d ? b ? \mathbf{t} : \mathbf{f} : \mathbf{f}$ $d ? c ? \mathbf{t} : \mathbf{f} : \mathbf{f}$ $d ? c ? b ? \mathbf{t} : \mathbf{f} : \mathbf{f} : \mathbf{f}$ $b ? \mathbf{t} : \mathbf{f}$ $c ? \mathbf{t} : \mathbf{f}$

Figure 8.4: Six BTT-FSM*s generated in less than 15 seconds.

The generated BTT-FSM*s are monitored most efficiently on RAM machines, due to the fact that conditional statements are implemented via jumps in memory. Monitoring BTT-FSM*s using rewriting does not seem appropriate because it would require linear time, as a function of number of

states, to extract the BTT associated to a state in a BTT-FSM*. However, we believe that the algorithm presented in Section 8.3 is satisfactory in practice if one is willing to use a rewriting engine for monitoring.

8.5 Conclusions

This paper presented a foundational study in using rewriting in runtime verification and monitoring of systems. After a short discussion on types of monitoring and mathematical and technological preliminaries, a finite trace linear temporal logic was defined, together with an immediate but inefficient implementation of a monitor following directly its semantics. Then an efficient but ineffective implementation based on dynamic programming was presented, which traverses the execution trace backwards. The first effective and relatively efficient rewriting algorithm was further introduced, based on the idea of transforming the monitoring requirements as events are received from the monitored program. A non-trivial improvement of this algorithm based on hashing rewriting results, thereby reducing the number of rewritings performed during trace analysis, was also proposed. The hashing corresponds to building an observer automaton on-the-fly, having the advantage that only the part of the automaton that is needed for analyzing a given trace is generated. The resulting algorithm is very efficient. Since in many cases one would want to generate an observer finite state machine (or automaton) a priori, for example when a rewriting system cannot be used for monitoring, or when minimal runtime overhead is needed, a specialized data-structure called a binary transition tree (BTT) and corresponding finite state machines were introduced, and an algorithm for generating minimal such monitors from temporal formulae was discussed.

All algorithms were implemented in surprisingly few lines of Maude code, illustrating the strength of rewriting for this particular domain. In spite of the reduced size of the code, the implementations seem to be efficient for practical purposes. As a consequence, we have demonstrated how rewriting can be used not only to experiment with runtime monitoring logics, but also as an implementation language. As an example of future work is the extension of LTL with real-time constraints. Since Maude by itself provides a high-level specification language, one can argue that Maude in its entirety can be used for writing requirements. Further work will show whether this avenue is fruitful. Some of the discussed results and algorithms have been already used in two NASA applications, JPAX and X9, but an extensive

experimental assesment of these techniques is left as future work.

Exercises

Exercise 13 *As seen in Chapters 8 and 9, there are different semantics for LTL. This can be quite confusing in practice, because one can write a safety property using LTL thinking of one semantics, say finite-trace semantics, but then use the monitor generation algorithm for the other semantics, say infinite-trace. (1) Give an LTL formula which is valid under one semantics but not under the other. (2) Give an LTL formula whose languages of bad-prefixes are different under the two semantics. (3) Describe an implementation that automatically checks whether a given formula admits different bad prefixes under finite-trace vs. infinite-trace semantics.*

Chapter 9

Efficient Monitoring of ω -Languages

currently from CAV'05 paper [84]; this material will eventually be dissolved in other parts of the book.

Abstract: We present a technique for generating efficient monitors for ω -regular-languages. We show how Büchi automata can be reduced in size and transformed into special, statistically optimal nondeterministic finite state machines, called *binary transition tree finite state machines (BTT-FSMs)*, which recognize precisely the minimal bad prefixes of the original ω -regular-language. The presented technique is implemented as part of a larger monitoring framework and is available for download.

9.1 Introduction

There is increasing recent interest in the area of *runtime verification* [137, 252], which is an area which aims at bridging testing and formal verification. In runtime verification, monitors are generated from system requirements. These monitors observe online executions of programs and check them against requirements. The checks can be either *precise*, with the purpose of detecting existing errors in the observed execution trace, or *predictive*, with the purpose of detecting errors that have not occurred in the observed execution but were “close to happening” and could possibly occur in other executions of the (typically concurrent) system. Runtime verification can be used either during

testing, to catch errors, or during operation, to detect and recover from errors. Since monitoring unavoidably adds runtime overhead to a monitored program, an important technical challenge in runtime verification is that of synthesizing *efficient* monitors from specifications.

Requirements of systems can be expressed in a variety of formalisms, not all of them necessarily easily monitorable. As perhaps best shown by the immense success of programming languages like Perl and Python, regular patterns can be easily devised and understood by ordinary software developers. ω -regular-languages [47, 268] add infinite repetitions to regular languages, thus allowing one to specify properties of reactive systems [188]. The usual acceptance condition in finite state machines (FSM) needs to be modified in order to recognize infinite words, thus leading to Büchi automata [68]. Logics like linear temporal logics (LTL) [188] often provide a more intuitive and compact means to specify system requirements than ω -regular patterns. It is therefore not surprising that a large amount of work has been dedicated to generating (small) Büchi automata from, and verifying programs against, LTL formulae [109, 268, 96, 107].

Based on the belief that ω -languages represent a powerful and convenient formalism to express requirements of systems, we address the problem of generating efficient monitors from ω -languages expressed as Büchi automata. More precisely, we generate monitors that recognize the *minimal bad prefixes* [175] of such languages. A bad prefix is a *finite* sequence of events which cannot be the prefix of any accepting trace. A bad prefix is minimal if it does not contain any other bad prefix. Therefore, our goal is to develop efficient techniques that read events of the monitored program incrementally, and precisely detect when a *bad prefix* has occurred. Dual to the notion of bad prefix is that of a good prefix, meaning that the trace will be accepted for any infinite extension of the prefix.

We present a technique that transforms a Büchi automaton into a special (nondeterministic) finite state machine, called a *binary transition tree finite state machine (BTT-FSM)*, that can be used as a monitor: by maintaining a set of possible states which is updated as events are available. A sequence of events is a bad prefix iff the set of states in the monitor becomes empty. One interesting aspect of the generated monitors is that they may contain a special state, called *neverViolate*, which, once reached, indicates that the specification is *not monitorable* from that moment on. That can mean either that the specification has been fulfilled (e.g., a specification $\diamond(x > 0)$ becomes fulfilled when x is first seen larger than 0), or that from that moment

on, there will always be some possible continuation of the execution trace. For example, the monitor generated for $\Box(a \rightarrow \Diamond b)$ will have exactly one state, *neverViolate*, reflecting the intuition that liveness properties cannot be monitored.

As usual, a program state is abstracted as a set of relevant atomic predicates that hold in that state. However, in the context of monitoring, the evaluation of these atomic predicates can be the most expensive part of the entire monitoring process. One predicate, for example, can say whether the vector $v[1..1000]$ is sorted. Assuming that each atomic predicate has a given evaluation cost and a given probability to hold, which can be estimated apriori either by static or by dynamic analysis, the BTT-FSM generated from a Büchi automaton executes a “conditional program”, called a *binary transition tree (BTT)*, evaluating atomic predicates *by need* in each state in order to statistically optimize the decision to which states to transit. One such BTT is shown in Fig. 6.4.

The work presented in this paper is part of a larger project focusing on *monitoring-oriented programming (MOP)* [54, 53] which is a tool-supported software development framework in which monitoring plays a foundational role. MOP aims at reducing the gap between specification and implementation by integrating the two through monitoring: specifications are checked against implementations at runtime, and recovery code is provided to be executed when specifications are violated. MOP is specification-formalism-independent: one can add one’s favorite or domain-specific requirements formalism via a generic notion of *logic plug-in*, which encapsulates a formal logical syntax plus a corresponding monitor synthesis algorithm. The work presented in this paper is implemented and provided as part of the LTL logic plugin in our MOP framework. It is also available for online evaluation and download on the MOP website [2].

Some Background and Related Work. Automata theoretic model-checking is a major application of Büchi automata. Many model-checkers, including most notably SPIN [144], use this technique. So a significant effort has been put into the construction of small Büchi automata from LTL formulae. Gerth *et al.* [109] show a tableaux procedure to generate on-the-fly Büchi automata of size $2^{O(|\varphi|)}$ from LTL formulae φ . Kesten *et al.* [163] describe a backtracking algorithm, also based on tableaux, to generate Büchi automata from formulae involving both past and future modalities (PTL), but no complexity results are shown. It is known that LTL model-checking is PSPACE-complete [250] and PTL is as expressive and as hard

as LTL [190], though exponentially more succinct [190]. Recently, Gastin and Oddoux [107] showed a procedure to generate standard Büchi automata of size $2^{O(|\varphi|)}$ from PTL via alternating automata. Several works [96, 109] describe simplifications to reduce the size of Büchi automata. Algebraic simplifications can also be applied *a priori* on the LTL formula. For instance, $a \mathcal{U} b \wedge c \mathcal{U} b \equiv (a \wedge c) \mathcal{U} b$ is a valid LTL congruence that will reduce the size of the generated Büchi automaton. All these techniques producing small automata are very useful in our monitoring context because the smaller the original Büchi automaton for the ω -language, the smaller the BTT-FSM. Simplifications of the automaton *with respect to monitoring* are the central subject of this paper.

Kupferman *et al.* [175] classify safety according to the notion of *informativeness*. Informative prefixes are those that “tell the whole story”: they witness the violation (or validation) of a specification. Unfortunately, not all bad prefixes are informative; e.g., the language denoted by $\Box(q \vee \circ(\Box(p))) \wedge \Box(r \vee \circ(\Box(\neg p)))$ does not include any word whose prefix is $\{q, r\}, \{q\}, \{!p\}$. This is a (minimal) bad but not informative prefix, since it does not witness the violation taking place in the next state. One can use the construction described in [175] to build an automaton of size $O(2^{2^{|\varphi|}})$ which recognizes all bad prefixes but, unfortunately, this automaton may be too large to be stored. Our fundamental construction is similar in spirit to theirs but we do not need to apply a subset construction on the input Büchi since we already maintain the set of possible states that the running program can be in. Geilen [108] shows how Büchi automata can be turned into monitors. The construction builds a tableaux similar to [109] in order to produce an FSM of size $O(2^{|\varphi|})$ for recognizing informative good prefixes. Here we detect all the minimal bad prefixes, rather than just the informative ones. Unlike model-checking where a user hopes to see a counter-example that witnesses the violation, when monitoring critical applications one might want to observe a problem as soon as it occurs.

The technique illustrated here is implemented as a plug-in in the MOP *runtime verification (RV)* framework [54, 53]. Other RV tools include JAVA-MAC [165], JPAX [126], JMPAX [242], and EAGLE [28]. JAVA-MAC uses a special interval temporal logic as the specification language, while JPAX and JMPAX support variants of LTL. These systems instrument the JAVA bytecode to emit events to an external monitor observer. JPAX was used to analyze NASA’s K9 Mars Rover code [19]. JMPAX extends JPAX with predictive capabilities. EAGLE is a finite-trace temporal logic and tool for

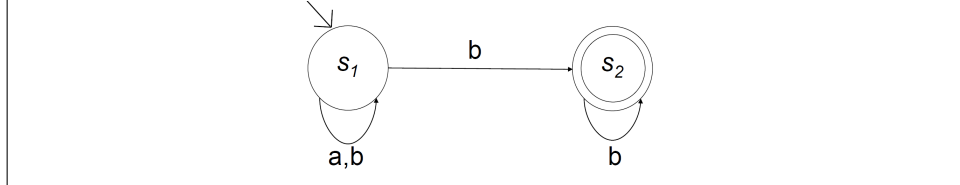


Figure 9.1: Büchi automaton recognizing the ω -regular expression $(a + b)^*b^\omega$

runtime verification, defining a logic similar to the μ -calculus with data-parameterization

9.2 Preliminaries: Büchi Automata

Büchi automata and their ω -languages have been studied extensively during the past decades. They are well suited to program verification because one can check satisfaction of properties represented as Büchi automata statically against transition systems [268, 68]. LTL is an important but proper subset of ω -languages.

Definition 39 A (nondeterministic) standard **Büchi automaton** is a tuple $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, where Σ is an **alphabet**, S is a set of **states**, $\delta: S \times \Sigma \rightarrow 2^S$ is a **transition function**, $S_0 \subseteq S$ is the set of **initial states**, and $\mathcal{F} \subseteq S$ is a set of **accepting states**.

In practice, Σ typically refers to events or actions in a system to be analyzed.

Definition 40 A Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ is said to **accept** an infinite word $\tau \in \Sigma^\omega$ iff there is some **accepting run** in the automaton, that is, a map $\rho: \text{Nat} \rightarrow S$ such that $\rho_0 \in S_0$, $\rho_{i+1} \in \delta(\rho_i, \tau_i)$ for all $i \geq 0$, and $\text{inf}(\rho) \cap \mathcal{F} \neq \emptyset$, where $\text{inf}(\rho)$ contains the states occurring infinitely often in ρ . The **language of** \mathcal{A} , $\mathcal{L}(\mathcal{A})$, consists of all words it accepts.

Therefore, ρ can be regarded as an infinite path in the automaton that starts with an initial state and contains at least one accepting state appearing infinitely often in the trace. Fig. 9.1 shows a nondeterministic Büchi automaton for the ω -regular expression $(a + b)^*b^\omega$ that contains all the infinite words over a and b with finitely many a s.

Definition 41 Let $\mathcal{L}(\mathcal{A})$ be the language of a Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$. A finite word $x \in \Sigma^*$ is a **bad prefix of \mathcal{A}** iff for any $y \in \Sigma^\omega$ the concatenation $xy \notin \mathcal{L}(\mathcal{A})$. A bad prefix is **minimal** if no other bad prefix is a prefix of it.

Therefore, no bad prefix of the language of a Büchi automaton can be extended to an accepted word. Similarly to [68], from now on we may tacitly assume that Σ is defined in terms of propositions over atoms. For instance, the self-transitions of s_1 in Fig. 9.1 can be represented as one self-transition, $a \vee b$.

9.3 Generating a *BTT-FSM* from a Büchi automaton

Not any property can be monitored. For example, in order to check a liveness property one needs to ensure that certain propositions hold infinitely often, which cannot be verified at runtime. This section describes how to transform a Büchi automaton into an efficient *BTT-FSM* that rejects precisely the minimal bad prefixes of the denoted ω -language.

Definition 42 A **monitor FSM (MFSM)** is a tuple $\langle \Sigma, S, \delta, S_0 \rangle$, where $\Sigma = P_A$ is an alphabet, S is a set of states potentially including a special state “*neverViolate*”, $\delta: S \times \Sigma \rightarrow 2^S$ is a transition function with $\delta(\text{neverViolate}, \text{true}) = \{\text{neverViolate}\}$ when $\text{neverViolate} \in S$, and $S_0 \subseteq S$ are initial states.

Note that we take Σ to be P_A , the set of propositions over atoms in A . Like *BTT-FSMs*, *MFSMs* may also have a special *neverViolate* state.

Definition 43 Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ be a sequence of transitions in the *MFSM* $\langle \Sigma, S, \delta, S_0 \rangle$, generated from $t = \theta_1 \theta_2 \dots \theta_j$, where $Q_0 = S_0$ and $Q_{i+1} = \bigcup_{s \in Q_i} \{\delta(s, \sigma) \mid \theta_{i+1} \models \sigma\}$, for all $0 \leq i < j$. We say that the *MFSM* **rejects** t iff $Q_j = \{\}$.

No finite extension of t will be **rejected** if $\text{neverViolate} \in Q_j$.

From Büchi to *MFSM*. We next describe two simplification procedures on a Büchi automaton that are sound w.r.t. monitoring, followed by the construction of an *MFSM*. The first procedure identifies segments of the

automaton which cannot lead to acceptance and can therefore be safely removed. As we will show shortly, this step is necessary in order to guarantee the soundness of the monitoring procedure. The second simplification identifies states with the property that if they are reached then the corresponding requirement cannot be violated by any *finite* extension of the trace, so monitoring is ineffective from there on. Note that reaching such a state does not necessarily mean that a good prefix has been recognized, but only that the property is *not monitorable* from there on.

Definition 44 Let $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be a Büchi automaton, C a connected component of its associated graph, and $\text{nodes}(C)$ the states associated to C . We say that C is **isolated** iff for any $s \in \text{nodes}(C)$ and $\sigma \in \Sigma$, it is the case that $\delta(s, \sigma) \subseteq \text{nodes}(C)$. We say that C is **total** iff for any $s \in \text{nodes}(C)$ and event θ , there are transitions σ such that $\theta \models \sigma$ and $\delta(s, \sigma) \cap \text{nodes}(C) \neq \emptyset$.

Therefore, there is no way to escape from an isolated connected component, and regardless of the upcoming event, it is always possible to transit from any node of a total connected component to another node in that component.

Removing Bad States. The next procedure removes states of the Büchi automaton which cannot be part of any accepting run (see Definition 2). Note that any state appearing in such an accepting run must eventually *reach* an accepting state. This procedure is fundamentally inspired by strongly-connected-component-analysis [163, 268], used to check emptiness of the language denoted by a Büchi automaton. Given a Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, let $U \subseteq S$ be the largest set of states such that the language of $\langle \Sigma, S, \delta, U, \mathcal{F} \rangle$ is empty. The states in U are unnecessary in \mathcal{A} , because they cannot change its language. Fortunately, U can be calculated effectively as the set of states that *cannot reach* any cycle in the graph associated to \mathcal{A} which contains at least one accepting state in \mathcal{F} . Fig. 9.2 shows an algorithm to do this.

The loop identifies maximal isolated connected components which do not contain any accepting states. The nodes in these components are marked as “bad”. The procedure `DFS_MARK_BAD` performs a depth-first-search in the graph and marks nodes as “bad” when all outgoing edges lead to a “bad” node. Finally, the procedure `REMOVE_BAD` removes all the bad states. The runtime complexity of this algorithm is dominated by the computation of maximal connected components. In our implementation, we used Tarjan’s $O(V + E)$ double DFS [68]. The proof of correctness is simple and it appears

```

INPUT : A Büchi automaton  $\mathcal{A}$ 
OUTPUT : A smaller Büchi automaton  $\mathcal{A}'$  such that  $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ .
REMOVE_BAD_STATES :
    for each maximal connected component  $C$  of  $\mathcal{A}$ 
        if ( $C$  is isolated and  $nodes(C) \cap \mathcal{F} = \emptyset$ ) then mark all states in  $C$  “bad”
    DFS_MARK_BAD ; REMOVE_BAD

```

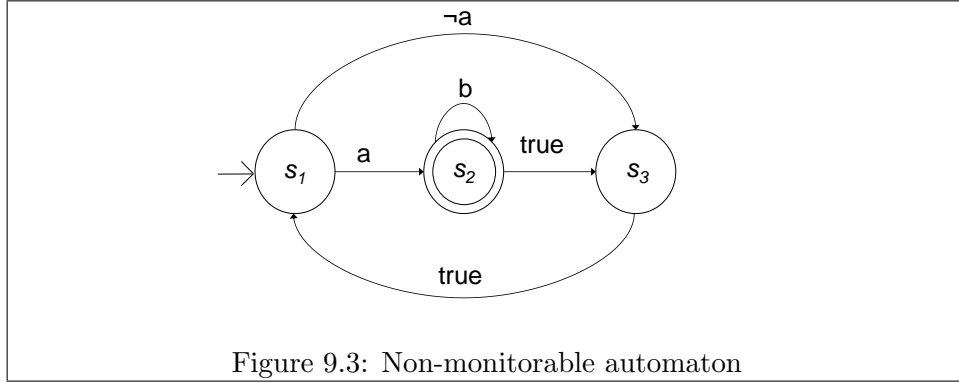
Figure 9.2: Removing bad states

in Appendix A. The Büchi automaton \mathcal{A}' produced by the algorithm in Fig. 9.2 has the property that there is some proper path from any of its states to some accepting state. One can readily generate an *MFSM* from a Büchi automaton \mathcal{A} by first applying the procedure **REMOVE_BAD_STATES** in Fig. 9.2, and then ignoring the acceptance conditions.

Theorem 18 *The *MFSM* generated from a Büchi automaton \mathcal{A} as above rejects precisely the minimal bad prefixes of $\mathcal{L}(\mathcal{A})$.*

Proof: Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the original Büchi automaton, let $\mathcal{A}' = \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ be the Büchi automaton obtained from \mathcal{A} by applying the algorithm in Fig. 9.2, and let $\langle \Sigma, S', \delta', S'_0 \rangle$ be the corresponding *MFSM* of \mathcal{A}' . For any finite trace $t = \theta_1 \dots \theta_j$, let us consider its corresponding sequence of transitions in the *MFSM* $Q_0 \xrightarrow{\theta_1} \dots \xrightarrow{\theta_j} Q_j$, where Q_0 is S'_0 . Note that the trace t can also be regarded as a sequence of letters in the alphabet Σ of \mathcal{A} , because we assumed Σ is P_A and because there is a bijection between propositions in P_A and A -events. All we need to show is that t is a bad prefix of \mathcal{A}' if and only if $Q_j = \emptyset$. Recall that \mathcal{A}' has the property that there is some non-empty path from any of its states to some accepting state. Thus, one can build an infinite path in \mathcal{A}' starting with any of its nodes, with the property that some accepting state occurs infinitely often. In other words, Q_j is not empty iff the finite trace t is the prefix of some infinite trace in $\mathcal{L}(\mathcal{A}')$. This is equivalent to saying that Q_j is empty iff the trace t is a bad prefix in \mathcal{A}' . Since Q_j empty implies $Q_{j'}$ empty for any $j > j'$, it follows that the *MFSM* rejects precisely the minimal bad prefixes of \mathcal{A} . \square

Theorem 1 says that the *MFSM* obtained from a Büchi automaton as above can be used as a monitor for the corresponding ω -language. Indeed, one only needs to maintain a current set of states Q , initially S'_0 , and transform it accordingly as new events θ are generated by the observed program: if $Q \xrightarrow{\theta} Q'$ then set Q to Q' ; if Q ever becomes empty then report



violation. Theorem 1 tells us that a violation will be reported as soon as a bad prefix is encountered.

Collapsing Never-Violate States. Reducing runtime overhead is crucial in runtime verification. There are many situations when the monitoring process can be safely stopped, because the observed finite trace cannot be finitely extended to any bad prefix. The following procedure identifies states in a Büchi automaton which cannot lead to the violation of any *finite* computation. For instance, the Büchi automaton in Fig. 9.3 can only reject infinite words in which the state s_2 occurs finitely many times; moreover, at least one transition is possible at any moment. Therefore, the associated *MFSM* will never report a violation, even though there are infinite words that are not accepted. We call such an automaton *non-monitorable*. This example makes it clear that if a state like s_1 is ever reached by the monitor, it does *not* mean that we found a good prefix, but that we could *stop looking* for bad prefixes.

Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be a Büchi automaton *simplified with REMOVE_BAD_STATES*. The procedure in Fig. 9.4 finds states which, if reached by a monitor, then the monitor can no longer detect violations regardless of what events will be observed in the future. The procedure first identifies the total connected components. According to the definition of totality, once a monitor reaches a state of a total connected component, the monitor will have the possibility to always transit within that connected component, thus never getting a chance to report violation. All states of a total component can therefore be marked as “never violate”. Other states can also be marked as such if, for any events, it is possible to transit from them to states already marked “never violate”; that is the reason for the disjunction in the second conditional. The procedure finds such nodes in a depth-first-search. Finally, COLLAPSE-NEVER_VIOLATE collapses all components marked “never violate”,

if any, to a distinguished node, *neverViolate*, having just a *true* transition to itself. If any collapsed node was in the initial set of states, then the entire automaton is collapsed to *neverViolate*. The procedure **GENERATE_MFSM** produces an *MFSM* by ignoring accepting conditions.

```

INPUT  : A Büchi automaton  $\mathcal{A}$ , cost function  $\varsigma$ , and probability function  $\pi$ .
OUTPUT : An effective BTT-FSM monitor rejecting the bad prefixes of  $\mathcal{L}(\mathcal{A})$ .
COLLAPSE_NEVER_VIOLATE :
  for each maximal connected component  $C$  of  $\mathcal{A}$ 
    if (  $C$  is total ) then mark all states in  $C$  as “never violate”
  for each  $s$  in depth-first-search visit
    if (  $\bigvee \{\sigma \mid \delta(s, \sigma) \text{ contains some state marked “never violate”} \}$  )
      then mark  $s$  as “never violate”
COLLAPSE-NEVER_VIOLATE ; GENERATE_MFSM ; GENERATE_BTT-FSM

```

Figure 9.4: Collapsing non-monitorable states

Taking as input this *MFSM*, say $\langle \Sigma, S', \delta', S'_0 \rangle$, cost function ς , and probability function π , **GENERATE_BTT-FSM** constructs a *BTT-FSM* $\langle A, S', btt, S'_0 \rangle$, where A corresponds to the set of atoms from which the alphabet Σ is built, and the map *btt*, here represented by a set of pairs, is defined as follows:

$$\begin{aligned}
btt = & \{ (neverViolate, \{neverViolate\}) \mid neverViolate \in S' \} \cup \\
& \{ (s, \beta_s) \mid s \in S' - \{neverViolate\} \wedge \beta_s \models \mu_s \}, \text{ where} \\
& \beta_s \text{ optimally implements } \mu_s \text{ w.r.t. } \varsigma \text{ and } \pi, \text{ with } \mu_s = \oplus (\bigcup \{ [\sigma : s'] \mid s' \in \delta'(s, \sigma) \})
\end{aligned}$$

The symbol \oplus denotes concatenation on a set of multi-transitions. Optimal *BTTs* β_s are generated like in Section 6.2. Proof of correctness appears in Appendix A.

9.4 Monitor Generation and MOP

We have shown that one can generate from a Büchi automaton a *BTT-FSM* recognizing precisely its bad prefixes. However, it is still necessary to integrate the *BTT-FSM* monitor within the program to be observed. *Monitoring-oriented programming* (MOP) [53] aims at merging specification and implementation through generation of runtime monitors from specifications and integration of those within implementation. In MOP, the task of generating monitors is divided into defining a logic engine and a language shell. The logic engine is concerned with the translation of specifications given as logical formulae into monitoring (pseudo-)code. The shell is responsible for the integration of the monitor within the application.

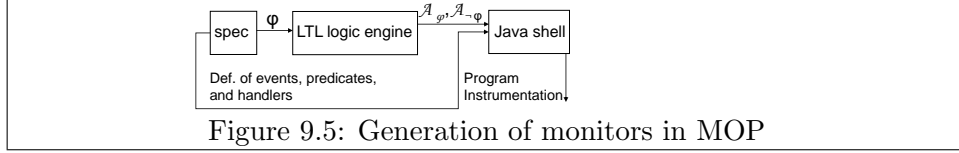


Fig. 9.5 captures the essence of the synthesis process of LTL monitors in MOP using the technique described in this paper. The user defines specifications either as annotations in the code or in a separate file. The specification contains definitions of events and state predicates, as well as LTL formulae expressing trace requirements. These formulae treat events and predicates as atomic propositions. Handlers are defined to track violation or validation of requirements. For instance, assume the events a and b denote the login and the logoff of the same user, respectively. Then the formula $\Box(a \rightarrow \circ(\neg a \mathcal{U} b))$ states that the user cannot be logged in more than once. A violation handler could be declared to track the user who logged in twice. The logic engine is responsible for the translation of the formulae φ and $\neg\varphi$ into two *BTT-FSM* monitors. One detects violation and the other validation of φ . Note that if the user is just interested in validation (no violation handler), then only the automaton for negation is generated. Finally, the language shell reads the definition of events and instruments the code so that the monitor will receive the expected notifications.

We used LTL2BA [208] to generate standard Büchi automata from LTL formulae. The described procedures are implemented in JAVA. This software and a WWW demo are available from the MOP website [2].

9.4.1 Evaluation

Table 1 shows *BTT-FSM* monitors for some LTL formulae. The *BTT* definition corresponding to a state follows the arrow (\rightsquigarrow). Initial states appear in brackets. For producing this table, we used the same cost and probabilities for all events and selected the smallest *BTT*. The first formula cannot be validated by monitoring and presents the permanent possibility to be violated; that is why its *BTT-FSM* does not have a *neverViolate* state. The second formula can never be violated since event a followed by event b can always occur in the future, so its *BTT-FSM* consists of just one state *neverViolate*. The last formula shows that our procedure does not aim at distinguishing validating from non-violating prefixes.

Table 2 shows that our technique can not only identify non-monitorable formulae, but also reduce the cost of monitoring by collapsing large parts of the Büchi automaton. We use the symbols \heartsuit , \clubsuit , and \spadesuit to denote,

Temporal Formula	<i>BTT-FSM</i>
$\Box(a \rightarrow b \mathcal{U} c)$	$[s_0] \rightsquigarrow c ? (b ? s_0 s_1 : s_0) : (a ? (b ? s_1 : \emptyset) : (b ? s_0 s_1 : s_0))$ $s_1 \rightsquigarrow b ? (c ? s_0 s_1 : s_1) : (c ? s_0 : \emptyset)$
$\Box(a \rightarrow \Diamond b)$	$[neverViolate] \rightsquigarrow \{neverViolate\}$
$a \mathcal{U} b \mathcal{U} c$	$[s_0] \rightsquigarrow c ? neverViolate : (a ? (b ? s_0 s_1 : s_0) : (b ? s_1 : \emptyset))$ $s_1 \rightsquigarrow c ? neverViolate : (b ? s_1 : \emptyset)$ $neverViolate \rightsquigarrow \{neverViolate\}$

Table 9.1: *BTT-FSMs* generated from temporal formulae

respectively, the effectiveness of `REMOVE_BAD_STATES`, the first, and the second loop of `COLLAPSE_NEVER_VIOLATE`. The first group contains non-monitorable formulae. The next contains formulae where monitor size could not be reduced by our procedures. The third group shows formulae where our simplifications could significantly reduce the monitor size. The last group shows examples of “accidentally” safe and “pathologically” safe formulae from [175]. A formula φ is accidentally safe iff not all bad prefixes are “informative” [175] (i.e., can serve as a witness for violation) but all computations that violate φ have an informative bad prefix. A formula φ is pathologically safe if there is a computation that violates φ and has no informative bad prefix. Since we detect *all* minimal bad prefixes, informativeness does not play any role in our approach. Both formulae are monitorable. For the last formula, in particular, a minimal bad prefix will be detected as soon as the monitor observes a $\neg a$, having previously observed a $\neg b$. One can generate and visualize the *BTT-FSMs* of all these formulae, and many others, online at [2].

9.5 Conclusions

Not all properties a Büchi automaton can express are monitorable. This paper describes transformations that can be applied to extract the monitorable components of Büchi automata, reducing their size and the cost of runtime verification. The resulting automata are called *monitor finite state machines* (*MFSMs*). The presented algorithms have polynomial running time in the size of the original Büchi automata and have already been implemented. Another contribution of this paper is the definition and use of *binary transition trees* (*BTTs*) and corresponding finite state machines (*BTT-FSMs*), as












Temporal Formula	# states	# transitions	symplif.
$\diamond a$	2 , 1	3 , 1	
$a \mathcal{U} \diamond b$	3 , 1	5 , 1	
$\Box(a \wedge b \rightarrow \diamond c)$	2 , 1	4 , 1	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} (\diamond d)))$	2 , 1	3 , 1	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} \Box(d \rightarrow \diamond e)))$	5 , 1	15 , 1	
$\neg a \mathcal{U} (b \mathcal{U} (c \mathcal{U} \Box(d \rightarrow \diamond e)))$	12 , 1	51 , 1	
$\neg \diamond a$	1 , 1	1 , 1	
$\Box(a \rightarrow b \mathcal{U} c)$	2 , 2	4 , 4	
$a \mathcal{U} (b \mathcal{U} (c \mathcal{U} d))$	4 , 4	10 , 10	
$a \wedge \diamond(\diamond b) \wedge \diamond(\Box(e))$	5 , 4	11 , 6	
$a \wedge \diamond(\diamond b) \wedge \diamond(\diamond c) \wedge \diamond(\Box(e))$	9 , 6	29 , 12	
$a \wedge \diamond(\diamond b) \wedge \diamond(\diamond c) \wedge \diamond(\diamond d) \wedge \diamond(\Box(e))$	17 , 10	83 , 30	
$a \wedge \diamond(\neg(\Box(b \rightarrow c \mathcal{U} d))) \wedge \diamond(\Box(e))$	7 , 5	20 , 10	
$\Box(a \vee \diamond(\Box(c))) \wedge \Box(b \vee \diamond(\Box(\neg c)))$	3 , 3	5 , 5	
$(\Box(a \vee \diamond(\Box(c))) \wedge \Box(b \vee \diamond(\Box(\neg c)))) \vee \Box(a) \vee \Box(b)$	12 , 6	43 , 22	

Table 9.2: Number of states and transitions before and after monitoring simplifications

well as a translation from *MFSMs* to *BTT-FSMs*. These special-purpose state machines encode optimal evaluation paths of boolean propositions in transitions.

We used LTL2BA [208] to generate Büchi automata from LTL, and JAVA to implement the presented algorithms. Our algorithms, as well as a graphical HTML interface, are available at [2]. This work is motivated by, and is part of, a larger project aiming at promoting monitoring as a foundational principle in software development, called *monitoring-oriented programming* (MOP). In MOP, the user specifies formulae, atoms, cost and probabilities associated to atoms, as well as violation and validation handlers. Then all these are used to automatically generate monitors and integrate them within the application.

This work is concerned with monitoring *violations* of requirements. In the particular case of LTL, *validations* of formulae can also be checked using the same technique by monitoring the negation of the input formula. Further work includes implementing the algorithm defined in [107] for generating Büchi automata of size $2^{O(|\varphi|)}$ from PTL, combining multiple formulae in a single automaton as showed by Ezick [97] so as to reduce redundancy of proposition evaluations, and applying further (standard) NFA simplifications to *MFSM*.

Appendix A. Proof of Correctness

(*Correctness of REMOVE_BAD_STATES*) Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the input and $\mathcal{A}' = \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ the output Büchi automata of the procedure. We want to show that their denoted languages are the same. Let ρ be an accepting run of \mathcal{A} . Then ρ can be fragmented in $\rho' \rho''$ where ρ' is the prefix whose states appear only finitely many times in ρ . Each state in ρ'' is therefore reachable from any other and therefore must be in a connected component where some states are in \mathcal{F} . The converse is also true, i.e., any connected component reachable from the set of initial states having at least one state in \mathcal{F} generates accepting runs (from [68] pp. 129). It is then immediate to notice that no accepting run ρ contains states that can never reach a state in \mathcal{F} . It turns out that any state belonging to an isolated component with no accepting states can be trivially removed as well as any state that can *only* make transitions to others which never reach an accepting state. This reachability analysis is performed in a depth-first-search as usual. Since only states that will never appear in accepting runs of \mathcal{A} are removed, it follows that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

(*Correctness of COLLAPSE_NEVER_VIOLATE*) First, we need to show that the *MFSM* produced by `GENERATE_MFSM` still detects bad prefixes of $\mathcal{L}(\mathcal{A})$, where \mathcal{A} is the input Büchi automaton having the property that all states can reach accepting states. From the observation that all states that have been collapsed to *neverViolate* can reach an accepting state, the proof of the first part is similar to that of Theorem 1 and is omitted. So, the *MFSM* is still a monitor for the bad prefixes of $\mathcal{L}(\mathcal{A})$. In other words, this simplification is sound w.r.t. monitoring minimal bad prefixes. In addition to this, we need to show that the state *neverViolate* is reached *only if* violations of the requirement can no longer occur. We prove this second correctness criteria by a case analysis on the shape of the *MFSM* associated to the input Büchi. Each case corresponds to a loop in `COLLAPSE_NEVER_VIOLATE`.

(case 1) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_i} Q_j$ be a sequence of transitions on the trace $\theta_1 \theta_2 \dots \theta_j$ produced by the *MFSM* corresponding to the input Büchi automaton \mathcal{A} . Recall that we assume `REMOVE_BAD_STATES` is called before `COLLAPSE_NEVER_VIOLATE`. That is, all states in the input automaton reach some state in \mathcal{F} . If some node in a total connected component C of the

graph associated to \mathcal{A} belongs to Q_j then it is not possible for any finite trace with prefix $\theta_1\theta_2...\theta_j$ to be rejected by the corresponding *MFSM* since it is always possible to make a transition between nodes in C from the definition of totality. Those states in C can be marked as “never violate” meaning that they denote the special *neverViolate* state.

(case 2) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_j} Q_j$ be a sequence of transitions produced on the trace $\theta_1\theta_2...\theta_j$ by the *MFSM* defined as before. If any state belongs to Q_j with the property that for further events θ_{j+1} there exists a transition to a state marked “never violate”, then there must exist some state marked “never violate” in Q_{j+1} . Such states can be found by checking the validity of the disjunction of propositions labeling the edges of transitions to states marked “never violate”. Since the monitor will definitely reach a state marked “never violate” in the trace $\theta_1\theta_2...\theta_j\theta_{j+1}$, it is therefore safe to reach “never violate” in Q_j as well, since violations are impossible from event j on.

The states marked with the “never violate” label can therefore be collapsed, since a violation cannot occur if any of these states is reached. The *BTT-FSM* is generated from the *MFSM* according to the defined construction.

Appendix B. Complexity Results

We use the definition of Binary Decision Trees (BDTs) due to Garey [106]. As we will show next, Garey’s BDTs serve as a procedure to identify one among a set of possible objects characterizing some aspect of interest. Hyafil and Rivest [156] proved the Optimal BDT problem NP-complete. Moret [203] shows that these BDTs are a restricted form of decision trees similar to binary transition trees as defined here.

Let $X = \{x_1, \dots, x_n\}$ be a finite set of objects and $\mathcal{T} = \{T_1, \dots, T_t\}$ a finite set of tests. For each test T_i , $1 \leq i \leq t$, and object x_j , $1 \leq j \leq n$, we either have $T_i(x_j) = \text{true}$ or $T_i(x_j) = \text{false}$. A binary decision tree associates tests in the root and all other internal nodes, and associates objects in X to terminal nodes. The *optimal decision tree* problem is to determine whether there exists a decision tree with cost less than or equal to w which completely identifies each element in X , given \mathcal{T} and X . The cost of this tree is defined as $\sum_{x_i \in X} p(x_i)$, where $p(x_i)$ is the length of the path from the root to the terminal identifying x_i .

One might think of the objects in X as possible answers to a question. The tests in \mathcal{T} serve to prune the data set of answers. A decision tree defines possible sequences of questions to ask in order to give a unique answer among

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	
T_1	T	F	T	F	T	F	F	T_1
T_2	F	T	F	F	F	T	T	T_6
T_3	T	F	F	F	F	F	F	T_3
T_4	F	F	F	F	T	F	F	T_4
T_5	F	F	F	F	F	T	F	T_5
T_6	F	F	F	F	F	F	T	T_2

Figure 9.6: Table of tests and a possible BDT for this table

those in X . The table of Fig. 9.6 appears in [106] and denotes the set of tests available to use in some binary decision tree.

The definition above [106] and the NP-completeness proof for the Optimal BDT problem [156] assume the table provided as input is unambiguous and completely identifies the objects in X . That is, any object in X can be distinguished by applying some sequence of tests. The decision tree in Fig. 9.6 appears in [106] as an identification procedure for the tests and objects defined in the adjacent table. The subtree to the left of a node denotes objects whose answers to the test labeling that node are *true* and the result of all other tests applied in the path to the root are consistent. The right subtree is similar. This is similar to binary transition trees. However, it is worth noting that in order to identify x_j one does not need to test all T_i having $T_i(x_j) = \text{true}$. Observe this in particular for x_6 and x_7 . It means that decision trees prune the data set in each test they make. A decision can be made whenever it is possible to identify an object. For instance, no object but x_7 assigns *false* to T_1 and *true* to T_6 . So this is a sufficient condition to identify x_7 . Moret showed that the construction of optimal binary decision trees, similar to *BTTs* (where tests take the form of boolean proposition over a set of atoms) is an NP-hard [203] problem. That implies that our *BTT* problem is also NP-hard.

Exercises

Exercise 14 Consider the following three (infinite-trace) LTL formulae from Table 9.2: $\Box(a \wedge b \rightarrow \Diamond c)$, $\Box(a \rightarrow b \mathcal{U} c)$, and, respectively, $a \wedge \circ(\Diamond b) \wedge \Diamond(\Box(e))$. For each of them do the following: (1) give a Buchi automaton that has the same language; (2) give an MFSM corresponding to (1) that rejects precisely the minimal bad prefixes (see Theorem 18); (3) optimize the MFSM in (2) by collapsing the non-monitorable states (algorithm in Figure 9.4); (4) give the optimal BTT-FSM corresponding to (3).

Chapter 10

Efficient Monitoring of “Always Past” Temporal Safety

This chapter contains material from [139]. See also [138].

Abstract: The problem of testing whether a finite execution trace of events generated by an executing program violates a linear temporal logic (LTL) formula occurs naturally in runtime analysis of software. Two efficient algorithms for this problem are presented in this paper, both for checking safety formulae of the form “always P ”, where P is a past time LTL formula. The first algorithm is implemented by rewriting and the second synthesizes efficient code from formulae. Further optimizations of the second algorithm are suggested, reducing space and time consumption. Special operators suitable for writing succinct specifications are discussed and shown equivalent to the standard past time operators. This work is part of NASA’s PathExplorer project, the objective of which is to construct a flexible framework for efficient monitoring and analysis of program executions.

Like in future time LTL, but dually, we can have various ways to interpret $\circ\varphi$. The stationary interpretation appears to be different from the strong / weak interpretation. That is, strong reduces to weak and viceversa, but it is not clear how stationary reduces to any of these or to anything else.

10.1 Introduction

The work presented in this paper is part of a project at NASA Ames Research Center, called PathExplorer [127, 126, 132, 125, 218], that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to analyze the execution trace of a running program to detect errors. The errors that we are considering at this stage are multi-threading errors such as deadlocks and data races, and non-conformance with linear temporal logic specifications, which is the main focus of this paper.

Linear Temporal Logic (LTL) [212, 186, 188] is a logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being a typical example. LTL has been mainly used to specify properties of concurrent and interactive down-scaled models of real systems, so that fully formal correctness proofs could subsequently be carried out, for example using theorem provers or model checkers (see for example [145, 140, 133]). However, formal proof techniques are usually not scalable to real sized systems without a substantial effort to abstract the system more or less manually to a model which can be analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can directly model check source code, such as Java and C [134, 265, 86, 146, 78, 26, 211]. Stateless model checkers [112, 256] try to avoid the abstraction process by not storing states. Although these systems provide high confidence, they scale less well because most of their internal algorithms are exponential or worse.

Testing scales well, and is by far the most used technique in practice to validate software systems. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls of ad hoc testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to pay

in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goal is to provide tools that are completely automatic, implement very efficient algorithms and find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve fault tolerance. The idea is that the failure may trigger a recovery action in the monitored program.

The idea of using LTL in program testing is not new. It has already been pursued in commercial tools such as Temporal Rover (TR) [87], which stimulated our work in a major way. In TR, future and past time LTL properties are stated as formal comments within the program at chosen program points, like assertions. These formal comments are then, by a pre-processor, translated into code, which is inserted at the position of the comments, and executed whenever reached during program execution¹. The MaC tool [184] is another example of a runtime monitoring tool that has inspired this work. Here Java bytecode is automatically instrumented to generate events of interest during the execution. Of special interest is the temporal logic used in MaC, which can be classified as a past time interval logic convenient for expressing monitoring properties in a succinct way. A theoretical contribution in this paper is Theorem 19, which shows that the MaC temporal logic, together with 10 others, is equivalent to the standard past time temporal logic. The path exploration tool described in [119] uses a future time temporal logic formula to guide the execution of a program for debugging purposes. Hence, the role of a temporal logic formula is turned around from monitoring a trace to generation of a trace.

Past time LTL has been shown to have the same expressiveness as future time LTL [103]. However, past time LTL is exponentially more succinct than future time LTL [190]. For example, a property like *"every response should be preceded by a request"* can be easily stated in past time logic (reflecting directly the previous sentence), but the corresponding future time representation becomes *"it's not the case that (there is no request until (there is a response and no request))"*. Hence, past time LTL is more convenient for specifying certain properties, and is the focus of this paper.

We present two efficient monitoring algorithms for checking safety formulae of the form "always P ", where P is a past time LTL formula, one based

¹The implementation details of TR are not public.

on formula rewriting and one based on synthesizing efficient monitoring code from a formula. The rewriting-based algorithm illustrates how rewriting can be used to easily and elegantly define new logics for monitoring. This may be of interest when experimenting with logics, or if logics are domain specific and change with the application, or if one simply wants a small and elegant implementation. The synthesis-based algorithm, on the other hand, generates a very effective monitor for the particular past time logic, and focuses on efficiency. It is also better suited for generating code that can be inserted in the monitored program, in contrast to the rewriting approach, where a rewriting engine must be called by an external call.

The first algorithm is implemented by rewriting using Maude [72, 74, 75], an executable specification language whose main operational engine is based on term rewriting. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage in the development of PathExplorer, we have actually developed a general framework using Maude which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. The rewriting algorithm presented in this paper instantiate that framework to our logic of interest, past time LTL. The second algorithm presented in this paper is designed to be as efficient and specialized as possible, thus adding the minimum possible amount of runtime overhead. It essentially synthesizes a special purpose, efficient monitoring code from formulae, which is further compiled into an executable monitor. Further optimizations of the second algorithm are suggested, making each monitoring step typically run in time lower than the size of the monitored formula. Both algorithms are based on the fact that the semantics of past time LTL can be defined recursively in such a way that one only needs to look one step, or event, backwards in order to compute the new truth value of a formula and of its subformulae, thus allowing one to process and then discard the events as they are received from the instrumented program. Several special operators suitable for writing succinct monitoring safety specifications are introduced and shown semantically equivalent to the standard past time operators.

Section 10.2 gives a short description of the PathExplorer architecture, putting the presented work in context. Section 10.3 recalls past time LTL and introduces several monitoring operators together with their semantics, then discusses several past time logics and finally shows their equivalences. Section 10.4 first presents our rewriting-based framework for defining and executing new monitoring logics, and then shows how past time LTL fits into

Figure 10.1: Overview of the PaX observer.

this framework. Section 10.5 finally explains our monitor-synthesis algorithm, together with optimizations and two ways to implement it. Section 10.6 concludes the paper.

10.2 The PathExplorer Architecture

PathExplorer, PaX, is a flexible environment for monitoring and analyzing program executions. A program (or a set of programs) to be monitored, is supposed to be instrumented to emit execution events to an observer, which then examines the events and checks whether they satisfy certain user-given constraints. We first give an overview of the *observer* that monitors the event stream. Then we discuss how a program is instrumented for monitoring of temporal logic properties. The instrumentation presented is specialized to Java, but the principles carry over to any programming language.

10.2.1 The Observer

The constraints to be monitored can be of different kinds and defined in different languages. Each kind of constraint is represented by a module. Such a constraint module in principle implements a particular logic or program analysis algorithm. Currently there are modules for checking deadlock potentials, data race potentials, and for checking temporal logic formulae in different logics. Amongst the latter, several modules have been implemented for checking future time temporal logic, and the work presented in this paper is the basis for a module for checking past time logic formulae. In general, the user can program new constraint modules and in this manner extend PaX in an easy way.

The system is defined in a component-based way, based on a dataflow view, where components are put together using a “pipeline” operator, see Figure 10.1. The dataflow between any two components is a stream of events in simple text format, without any a-priori assumptions about the format of the events; the receiving component just ignores events it cannot recognize.

This simplifies composition and allows for components to be written in different languages and in particular to define observers of arbitrary systems, programmed in a variety of programming languages. This latter fact is important at NASA since several systems are written in a mixture of C, C++ and Java.

The central component of the PaX system is a so-called *dispatcher*. The dispatcher receives events from the executing program or system and then retransmits the event stream to each of the constraint modules. Each module is running in its own process with one input pipe, only dealing with events that are relevant to the module. For this purpose each module is equipped with an event parser. The dispatcher takes as input a configuration script, which specifies a list of commands - a command for each module that starts the module in a process. The dispatcher may read its input event stream from a file, or alternatively from a socket, to which the instrumented running program must write the event stream. In the latter case, monitoring can happen on-the-fly as the event stream is produced, and potentially on a different computer than the observed system.

10.2.2 Code Instrumentation

The program or system to be observed must be instrumented to emit execution events to the dispatcher (writing them to a file or to a socket as discussed above). We have currently implemented an automated instrumentation package for Java bytecode using the Java bytecode engineering tool JTrek [162]. The instrumentation package together with PathExplorer is called Java PathExplorer (JPaX). Given information about what kind of events to be emitted, the instrumentation package instruments the bytecode by inserting extra code for emitting events. For deadlock analysis, for example, events are generated that inform about lock acquisitions and releases. For temporal logic monitoring, one specifies the variables to be observed, and what predicates over these variables one wants to refer to in the temporal properties to be monitored. Imagine for example that the observer monitors the formula: “*always p*”, involving the predicate p , and that p is intended to be defined as $p \equiv x > y$, where x and y are static variables defined in a class C . In this case all updates to these variables must be instrumented, such that an update to any of them causes the predicate to be evaluated, and a toggle p to be emitted to the observer in case it has changed. The instrumentation script is written in Java (using reflection), but in essence can be represented as follows:

Figure 10.2: Events corresponding to observing predicate $p \equiv x > y$.

```
monitor C.x, C.y;
proposition p is C.x > C.y;
```

The code will then be instrumented to emit changes in the predicate p . More specifically, first the initial value of the predicate is transmitted to the observer. Subsequently, whenever one of the two variables is updated, the predicate is evaluated, and in case its value has changed since last evaluation, the predicate name p is transmitted to the observer as a toggle. The observer keeps track of the value of the predicate, based on its initial value, and the subsequent predicate toggles. Figure 10.2 shows an execution trace where x and y initially are 0, and then subsequently updated. The corresponding values of p are shown. Also shown are the events that are sent to the observer. That is, the initial value of p and the subsequent p toggles.

10.3 Finite Trace Past Time LTL

In this section we remind some basic notions of finite trace linear past time temporal logic [186, 188], establish some conventions and introduce some operators that we found particularly useful for runtime monitoring. We emphasize that the semantics of past time LTL can be elegantly defined recursively, thus allowing us to implement monitoring algorithms that only need to look one step backwards. We also show that past time LTL can be entirely defined using just the special operators, that were introduced essentially because of practical needs, thus strengthening our belief that past time LTL is an appropriate candidate logic for expressing monitoring safety requirements.

10.3.1 Syntax

We allow the following constructors for formulae, where A is a finite set of “atomic propositions”:

$$\begin{aligned}
 F ::= & \text{ true } \mid \text{ false } \mid A \mid \neg F \mid F \text{ op } F \\
 & \text{(propositional operators)} \\
 & \odot F \mid \diamond F \mid \Box F \mid F \mathcal{S}_s F \mid F \mathcal{S}_w F \\
 & \text{(standard past time operators)} \\
 & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w \\
 & \text{(monitoring operators)}
 \end{aligned}$$

The propositional binary operators, op , are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction.

The standard past time and the monitoring operators are often called “temporal operators”, because they refer to other (past) moments in time. The operator $\odot F$ should be read “previously F ”; its intuition is that F held at the immediately previous moment in time. $\diamond F$ should be read “eventually in the past F ”, with the intuition that there is some past moment in time when F was true. $\Box F$ should be read “always in the past F ”, with the obvious meaning. The operator $F_1 \mathcal{S}_s F_2$, which should be read “ F_1 strong since F_2 ”, reflects the intuition that F_2 held at some moment in the past and, since then, F_1 held all the time. $F_1 \mathcal{S}_w F_2$ is a weak version of “since”, read “ F_1 weak since F_2 ”, saying that either F_1 was true all the time or otherwise $F_1 \mathcal{S}_s F_2$.

The monitoring operators \uparrow , \downarrow , $[-, -)_s$, and $[-, -)_w$ were inspired by work in runtime verification in [184]. We found these operators often more intuitive and compact than the usual past time operators in specifying runtime requirements, despite the fact that they have the same expressive power as the standard ones, as we discovered later. The operator $\uparrow F$ should be read “start F ”; it says that the formula F just started to be true, that is, it was false previously but it is true now. Dually, the operator $\downarrow F$ which is read “end F ”, carries the intuition that F ends to be true, that is, it was previously true but it is false now. The operators $[F_1, F_2)_s$ and $[F_1, F_2)_w$ are read “strong/weak interval F_1, F_2 ” and they carry the intuition that F_1 was true at some point in the past but F_2 has not been seen to be true since then, including that moment. For example, if `START` and `DOWN` are predicates on the state of a web server to be monitored, then $[\text{START}, \text{DOWN})_s$ is a property stating that the server *was* rebooted recently and since then it was not down,

$t \models \text{true}$	is always true,
$t \models \text{false}$	is always false,
$t \models a$	iff $a(s_n)$ holds,
$t \models \neg F$	iff $t \not\models F$,
$t \models F_1 \text{ op } F_2$	iff $t \models F_1$ and/or/etc. $t \models F_2$, when op is \wedge/\vee /etc.,
$t \models \circ F$	iff $t' \models F$, where $t' = t_{n-1}$ if $n > 1$ and $t' = t$ if $n = 1$,
$t \models \diamond F$	iff $t_i \models F$ for some $1 \leq i \leq n$,
$t \models \Box F$	iff $t_i \models F$ for all $1 \leq i \leq n$,
$t \models F_1 \mathcal{S}_s F_2$	iff $t_j \models F_2$ for some $1 \leq j \leq n$ and $t_i \models F_1$ for all $j < i \leq n$,
$t \models F_1 \mathcal{S}_w F_2$	iff $t \models F_1 \mathcal{S}_s F_2$ or $t \models \Box F_1$,
$t \models \uparrow F$	iff $t \models F$ and $t_{n-1} \not\models F$,
$t \models \downarrow F$	iff $t_{n-1} \models F$ and $t \not\models F$,
$t \models [F_1, F_2]_s$	iff $t_j \models F_1$ for some $1 \leq j \leq n$ and $t_i \not\models F_2$ for all $j \leq i \leq n$,
$t \models [F_1, F_2]_w$	iff $t \models [F_1, F_2]_s$ or $t \models \Box \neg F_2$.

Figure 10.3: Semantics of finite trace past time LTL.

while $[\text{START}, \text{DOWN}]_w$ says that the server was not down recently, meaning that it was either not down at all recently or it was rebooted and since then it was not down.

10.3.2 Formal Semantics

We next present formally the intuitive semantics described above. We regard a trace as a finite sequence of abstract states. In practice, these states are generated by events emitted by the program or system that we want to observe. Such events could indicate when variables' values are changed or when locks are acquired or released by threads or processes, or even when a physical action takes place, such as opening or closing a valve, a gate, or a door. If s is a state and a is an atomic proposition then $a(s)$ is true if and only if a holds in the state s . Notice that we are loose with respect to what “holds” means, because, depending on the context, it can mean anything. However, in the case of JPAX the atomic predicates are just any Java boolean expressions and their satisfaction is decided by evaluating them in the current state of the Java program. If $t = s_1 s_2 \dots s_n$ ($n \geq 1$) is a trace then we let t_i denote the trace $s_1 s_2 \dots s_i$ for each $1 \leq i \leq n$. The formal semantics of the operators defined in the previous subsection is given in Figure 10.3.

$$\begin{array}{ll}
t \models \Diamond F & \text{iff } t \models F \text{ or } (n > 1 \text{ and } t_{n-1} \models \Diamond F), \\
t \models \Box F & \text{iff } t \models F \text{ and } (n > 1 \text{ implies } t_{n-1} \models \Box F), \\
t \models F_1 \mathcal{S}_s F_2 & \text{iff } t \models F_2 \text{ or } (n > 1 \text{ and } t \models F_1 \text{ and } t_{n-1} \models F_1 \mathcal{S}_s F_2), \\
t \models F_1 \mathcal{S}_w F_2 & \text{iff } t \models F_2 \text{ or } (t \models F_1 \text{ and } (n > 1 \text{ implies } t_{n-1} \models F_1 \mathcal{S}_w F_2)), \\
t \models [F_1, F_2]_s & \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } (n > 1 \text{ and } t_{n-1} \models [F_1, F_2]_s)), \\
t \models [F_1, F_2]_w & \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } (n > 1 \text{ implies } t_{n-1} \models [F_1, F_2]_w)).
\end{array}$$

Figure 10.4: Recursive semantics of finite trace past time LTL.

Notice the special semantics of the operator “previously ” on a trace of one state: $s \models \circ F$ iff $s \models F$. This is consistent with the view that a trace consisting of exactly one state s is considered like a *stationary* infinite trace containing only the state s . We adopted this view because of intuitions related to monitoring. One can start monitoring a process potentially at any moment, so the first state in the trace might be different from the initial state of the monitored process. We think that the “best guess” one can have w.r.t. the past of the monitored program is that it was stationary. Alternatively, one could consider that $\circ F$ is false on a trace of one state for any atomic proposition F , but we find this semantics inconvenient because some atomic propositions may be related, such as, for example, a proposition “gate-up” and a proposition “gate-down”.

10.3.3 Recursive Semantics

An observation of crucial importance in the design of the subsequent algorithms is that the semantics above can be defined recursively, in such a way that the satisfaction relation for a formula and a trace can be calculated along the execution trace looking only one step backwards, as shown in Figure 10.4.

For example, according to the formal, nonrecursive, semantics, a trace $t = s_1 s_2 \dots s_n$ satisfies the formula $[F_1, F_2]_w$ if and only if either F_2 was false all the time in the past or otherwise F_1 was true at some point and since then F_2 was always false, including that moment. Therefore, in the case of a trace of size 1, i.e., when $n = 1$, it follows immediately that $t \models [F_1, F_2]_w$ if and only if $t \not\models F_2$. Otherwise, if the trace has more than one event then first of all $t \not\models F_2$, and then either $t \models F_1$ or else the prefix trace satisfies the interval formula, that is, $t_{n-1} \models [F_1, F_2]_w$. Similar reasoning applies to the other recurrences.

10.3.4 Equivalent Logics

We call the past time temporal logic presented above $ptLTL$. There is a tendency among logicians to minimize the number of operators in a given logic. For example, it is known that two operators are sufficient in propositional calculus, and two more (“next” and “until”) are needed for future time temporal logics. There are also various ways to minimize $ptLTL$. Let $ptLTL|_{Ops}$ be the restriction of $ptLTL$ to the propositional operators plus the operations in Ops . Then

Theorem 19 *The following 12 logics are all equivalent to $ptLTL$:*

1. $ptLTL|_{\{\ominus, \mathcal{S}_s\}}$,
2. $ptLTL|_{\{\ominus, \mathcal{S}_w\}}$,
3. $ptLTL|_{\{\ominus, []_s\}}$,
4. $ptLTL|_{\{\ominus, []_w\}}$,
5. $ptLTL|_{\{\uparrow, \mathcal{S}_s\}}$,
6. $ptLTL|_{\{\uparrow, \mathcal{S}_w\}}$,
7. $ptLTL|_{\{\uparrow, []_s\}}$,
8. $ptLTL|_{\{\uparrow, []_w\}}$,
9. $ptLTL|_{\{\downarrow, \mathcal{S}_s\}}$,
10. $ptLTL|_{\{\downarrow, \mathcal{S}_w\}}$,
11. $ptLTL|_{\{\downarrow, []_s\}}$,
12. $ptLTL|_{\{\downarrow, []_w\}}$.

The first two are known in the literature [186].

Proof: We first show the following properties:

1.	$\diamond F$	$=$	$true \mathcal{S}_s F$
2.	$\Box F$	$=$	$\neg \diamond \neg F$
3.	$F_1 \mathcal{S}_w F_2$	$=$	$(\Box F_1) \vee (F_1 \mathcal{S}_s F_2)$
4.	$\Box F$	$=$	$F \mathcal{S}_w false$
5.	$\diamond F$	$=$	$\neg \Box \neg F$
6.	$F_1 \mathcal{S}_s F_2$	$=$	$(\diamond F_2) \wedge (F_1 \mathcal{S}_w F_2)$
7.	$\uparrow F$	$=$	$F \wedge \neg \circ F$
8.	$\downarrow F$	$=$	$\neg F \wedge \circ F$
9.	$[F_1, F_2]_s$	$=$	$\neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_s F_1)$
10.	$[F_1, F_2]_w$	$=$	$\neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1)$
11.	$\downarrow F$	$=$	$\uparrow \neg F$
12.	$\uparrow F$	$=$	$\downarrow \neg F$
13.	$[F_1, F_2]_w$	$=$	$(\Box \neg F_2) \vee [F_1, F_2]_s$
14.	$[F_1, F_2]_s$	$=$	$(\diamond F_1) \wedge [F_1, F_2]_w$
15.	$\circ F$	$=$	$(F \rightarrow \neg \uparrow F) \wedge (\neg F \rightarrow \downarrow F)$
16.	$F_1 \mathcal{S}_s F_2$	$=$	$F_2 \vee [\circ F_2, \neg F_1]_s$

These properties are intuitive and relatively easy to prove. For example, property 15., the definition of $\circ F$ in terms of $\uparrow F$ and $\downarrow F$, says that in order to find out the value of a formula F in the previous state it suffices to look at the value of the formula in the current state and then, if it is true then look if the formula just started to be true or else look if the formula just ended to be true. We next only prove property 10., the proofs of the others are similar and straightforward.

In order to prove 10., one needs to show that for any trace t , it is the case that $t \models [F_1, F_2]_w$ if and only if $t \models_w \neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1)$. We show this by induction on the size of the trace t . If the size of t is 1, that is, if $t = s_1$, then

$$\begin{aligned}
t \models [F_1, F_2]_w &\text{ iff} \\
&\text{ iff } t \not\models F_2 \\
&\text{ iff } t \models \neg F_2 \\
&\text{ iff (by "absorption" in boolean reasoning)} \\
&\quad t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \neg F_2) \\
&\text{ iff } t \models \neg F_2 \text{ and } (t \models F_1 \text{ or } t \models \circ \neg F_2) \\
&\text{ iff } t \models \neg F_2 \wedge ((\circ \neg F_2) \mathcal{S}_w F_1).
\end{aligned}$$

If the size of the trace t is $n > 1$ then

$$\begin{aligned}
t \models [F_1, F_2]_w & \text{ iff} \\
& \text{iff (by the recursive semantics)} \\
& t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t_{n-1} \models [F_1, F_2]_w) \\
& \text{iff (by the induction hypothesis)} \\
& t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t_{n-1} \models \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1)) \\
& \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t_{n-1} \models \neg F_2 \text{ and } t_{n-1} \models (\odot \neg F_2) \mathcal{S}_w F_1) \\
& \text{iff } t \not\models F_2 \text{ and } (t \models F_1 \text{ or } t \models \odot \neg F_2 \text{ and } t_{n-1} \models (\odot \neg F_2) \mathcal{S}_w F_1) \\
& \text{iff (by the recursive semantics)} \\
& t \not\models F_2 \text{ and } t \models (\odot \neg F_2) \mathcal{S}_w F_1 \\
& \text{iff } t \models \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1).
\end{aligned}$$

Therefore, $[F_1, F_2]_w = \neg F_2 \wedge ((\odot \neg F_2) \mathcal{S}_w F_1)$.

The equivalences of the 12 logics with *ptLTL* follow now immediately. For example, in order to show the 8th logic, $ptLTL|_{\{\uparrow, \downarrow_s\}}$, equivalent to *ptLTL*, one needs to show how the operators \uparrow and $[-, -]_s$ can define all the other past time temporal operators. This is straightforward because, 11. shows how \downarrow can be defined in terms of \uparrow , 15. shows how $\odot F$ can be defined using just \uparrow and \downarrow , 16. defines \mathcal{S}_s , 1. defines \diamond , 2. \Box , 3. \mathcal{S}_w , and 13. defines the weak interval. The interested reader can check the other 11 equivalences of logics. \square

Unlike in theoretical research, in practical monitoring of programs we want to have as many temporal operators available as possible and *not* to automatically translate them into a reduced kernel set. The reason is twofold. On the one hand, the more operators are available, the more succinct and natural the task of writing requirement specifications. On the other hand, as seen later in the paper, additional memory is needed for each temporal operator, so we want to keep the formulae as concise as possible.

10.4 Monitoring Safety by Rewriting

The architecture of JPAX is such that events extracted from a running program are sent to an observer which decides whether requirements are violated or not. An important concern that we had and are still having at this

relatively incipient stage of JPAX, is whether the chosen monitoring logics are expressive enough to specify powerful, practical and interesting requirements. Since flexibility with respect to defining/modifying monitoring logics is a very important factor at this stage, we have developed a rewriting-based framework which allows one to easily and effectively define new logics for runtime analysis and to monitor execution traces against formulae in these logics. We use the rewriting system Maude as a basis for this framework. In the following we first present Maude, and how formulae and important data structures are represented in Maude. Then we describe how basic propositional calculus is defined in Maude, and then how *ptLTL* is defined. Finally, it is described how this Maude definition of *ptLTL* is used for monitoring requirements stated by the user on execution traces.

10.4.1 Maude

We have implemented our logic-defining framework by rewriting in Maude [72, 74, 75]. Maude is a modularized membership equational [201] and rewriting logic [200] specification and verification system, whose operational engine is mainly based on a very efficient implementation of rewriting. A Maude module consists of sort and operator declarations, as well as equations relating terms over the operators and universally quantified variables. Modules can be composed in a hierarchical manner, building new theories from old theories. A particular attractive aspect of Maude is its *mix-fix* notation for syntax, which, together with precedence attributes of operators gives us an elegant way to compactly define syntax of logics. For example, the operation declarations²:

```

op _/\_ : Expression Expression
    -> Expression [prec 33] .
op _\/_ : Expression Expression
    -> Expression [prec 40] .
op [_,_] : Expression Expression
    -> Expression .
op if_then_else_ : Bool Expression Expression
    -> Expression .

```

define a simple syntax over the sort *Expression*, where conjunction and disjunction are infix operators (the underscores stand for arguments, whose sorts are listed after the colon), while the interval and the conditional are mix-fix: operator and arguments can be mixed. Conjunction binds tighter than

²These declarations are artificial and intended to explain some of Maude's features; they will not be needed later in the paper.

disjunction because it has a lower precedence (the lower the precedence the tighter the binding), so one is relieved from having to add useless parentheses to one's formulae.

It is often the case that equational and/or rewriting logics act like foundational logics, in the sense that other logics, or more precisely their syntax and operational semantics, can be expressed and efficiently executed by rewriting, so we regard Maude as a good choice to develop and prototype with various monitoring logics. The Maude implementations of the current logics supported by JPAX are quite compact. They are based on a simple, general architecture to define new logics which we only describe informally in the next subsection. Maude's notation will be introduced "on the fly" as needed.

10.4.2 Formulae and Data Structures

We have defined a generic module, called `FORMULA`, which defines the infrastructure for all the user-defined logics. Its Maude code is rather technical and so will not be given here. The module `FORMULA` includes some designated basic sorts, such as `Formula` for syntactic formulae, `FormulaDS` for formula data structures needed when more information than the formula itself should be stored for the next transition as in the case of past time LTL, `Atom` for atomic propositions (or state variables), `AtomState` for assignments of boolean values to atoms, also called "states", and `AtomState*` for such assignments together with *final* assignments, i.e., those that are followed by the end of a trace, sometimes requiring a special evaluation procedure. A state `As` is made terminal by applying it a unary operator, `_* : AtomState -> AtomState*`. `Formula` is a subsort of `FormulaDS`, because there are logics in which no extra information but a modified formula needs to be carried over for the next iteration (such as future time LTL which is also provided by JPAX). There are two constants of sort `Formula` provided, namely `true` and `false`, with the obvious meaning. The propositions that hold in a certain program state are generated by the executing instrumented program.

One of the most important operators in `FORMULA` is `_{}:FormulaDS AtomState* -> FormulaDS`, which updates the formula data structure when an (abstract) state change occurs during the execution of the program. Notice the use of mix-fix notation for operator declaration, in which underscores represent places of arguments, their order being the one in the arity of the operator. On atomic propositions, say `A`, the module `FORMULA` defines the "update" operator as follows: `A{As*}` is true or false, depending on whether

As^* assigns true or false to the atom A , where As^* is an atom state (i.e., an assignment from atoms to boolean values), which is either a terminal state (the last in a trace) or not. In the case of propositional calculus, this update operation basically evaluates propositions in the new state. For other logics it can be more complicated, depending on their trace semantics.

10.4.3 Propositional Calculus

Propositional calculus should be included in any monitoring logic. Therefore, we begin with the following module which is heavily used in JPAX. It implements an efficient rewriting procedure due to Hsiang [151] to decide validity of propositions, reducing any boolean expression to an exclusive disjunction (formally written $_{++}$) of conjunctions ($_{/\backslash}$):

```
fmod PROP-CALC is extending FORMULA .
*** Constructors ***
  op _/\_ : Formula Formula
          -> Formula [assoc comm] .
  op _++_ : Formula Formula
          -> Formula [assoc comm] .

  vars X Y Z : Formula . var As* : AtomState* .
  eq true /\ X = X .
  eq false /\ X = false .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ X = X .
  eq X /\ (Y ++ Z) = (X /\ Y) ++ (X /\ Z) .

*** Derived operators ***
  op _\/_ : Formula Formula -> Formula .
  op _->_ : Formula Formula -> Formula .
  op _<->_ : Formula Formula -> Formula .
  op !_ : Formula -> Formula .
  eq X \/_ Y = (X /\ Y) ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ (X /\ Y) .
  eq X <-> Y = true ++ X ++ Y .

*** Operational Semantics
  eq (X /\ Y){As*} = X{As*} /\ Y{As*} .
  eq (X ++ Y){As*} = X{As*} ++ Y{As*}
endfm
```

In Maude, operators are introduced after the `op` and `ops` (when more than one operator is introduced) symbols. Operators can be given attributes in square brackets, such as associativity and commutativity. Universally quantified variables used in equations are introduced after the `var` and `vars` symbols. Finally, equations are introduced after the `eq` symbol. The specification of the simple propositional calculus above shows the flexibility of the mix-fix notation of Maude, which allows us to define the syntax of a logic in the most natural way.

The equations above are interpreted as rewriting rules by Maude, so they will be applied from left to right only. However, due to the associativity and commutativity attributes, rewrites as well as matchings are applied *modulo* associativity and commutativity (AC), making therefore the procedure implied by the rewrite rules for propositional calculus above highly non-trivial. As proved by Hsiang [151], the AC rewriting system above has the property that any proposition is reduced to true or false if it is semantically true or false, or otherwise to a canonical form modulo AC; thus two formulae are equivalent if and only if their canonical forms are equal modulo AC. We found this procedure quite convenient so far, being able to efficiently reduce formulae of hundreds of symbols that occurred in practical examples. However, one should of course not expect this procedure to work efficiently on any proposition, because the propositional validity problem is NP-complete.

10.4.4 Past Time Linear Temporal Logic

Past time LTL can now be implemented on top of the provided logic-defining framework. Our rewriting based implementation below follows the recursive semantics of past time LTL defined in Subsection 10.3.3, and, it appears similar to the Java implementation used in [184]. We next explain the PT-LTL module in detail.

We start by defining the syntax of past time LTL. Since it extends the module PROP-CALC of propositional calculus, we only have to define syntax for the temporal operators:

```
fmod PT-LTL is extending PROP-CALC .
  op (*)_ : Formula -> Formula .
                                     *** previously
  op <*_ : Formula -> Formula .
                                     *** eventually in the past
  op [*]_ : Formula -> Formula .
                                     *** always in the past
```

```

op _Ss_ : Formula Formula -> Formula .
      *** strong since
op _Sw_ : Formula Formula -> Formula .
      *** weak since
op start : Formula -> Formula .
      *** start
op end   : Formula -> Formula .
      *** end
op [_,_]s : Formula Formula -> Formula .
      *** strong interval
op [_,_]w : Formula Formula -> Formula .
      *** weak interval

```

We have used a curly bracket to close the intervals because, for some technical parsing related reasons, Maude does not allow unbalanced parentheses in its terms. The syntax above can now be used by users to write monitoring requirements as formulae. These formulae are loaded by JPAX at initialization and then sent to Maude for parsing and processing. When the first event from the instrumented program is received by JPAX, it sends this event to Maude in order to initialize its monitoring data structures associated to its formulae (remember that the recursive definition of past time LTL in Subsection 10.3.3 treats the first event of the trace differently). This is done by launching the reduction `mkDS(F, As)` in Maude, where `F` is the formula to monitor and `As` is the atom state abstracting the first event generated by the monitored program; `mkDS` is an abbreviation for “make data structure” and is defined below.

Before we define the operation `mkDS`, we first discuss the formula data structures storing not only the formulae but also their current satisfaction status. It is worth noticing that the strong and weak temporal operators have exactly the same recursive semantics starting with the second event. That suggests that we do not need nodes of different type (strong and weak) in the formula data structure once the monitoring process is initialized: the difference between strong and weak versions of an operator are rather represented by the initial values passed as arguments to a single common version of the operator. The following operation declarations therefore define the constructors for these data structures:

```

op atom          : Atom Bool -> FormulaDS .
op and   :   FormulaDS FormulaDS Bool -> FormulaDS .
op xor   :   FormulaDS FormulaDS Bool -> FormulaDS .
op previously : FormulaDS Bool -> FormulaDS .

```

```

op eventuallyPast    : FormulaDS Bool -> FormulaDS .
op alwaysPast       : FormulaDS Bool -> FormulaDS .
op since            : FormulaDS FormulaDS Bool -> FormulaDS .
op start            : FormulaDS Bool -> FormulaDS .
op end              : FormulaDS Bool -> FormulaDS .
op interval         : FormulaDS FormulaDS Bool -> FormulaDS .

```

The first operation defines a cell storing an atomic proposition together with its observed boolean value, while the next two store conjunction and exclusive disjunction nodes. According to the propositional calculus procedure defined in module PROP-CALC in Subsection 10.4.3, these are the only propositional operators that can occur in reduced formulae. The remaining operators are the seven past time temporal operators introduced so far.

An operator that extracts the boolean value associated to a temporal formula is needed in the sequel, so we define it next. The syntax of this operator is `[_] : FormulaDS -> Bool` and it is defined in the module FORMULA, together with its obvious equations `[true] = true` and `[false] = false`. Its definition on temporal and propositional and temporal operators follows:

```

var A : Atom . var B : Bool .
vars D Dx Dy : FormulaDS .
eq [and(Dx,Dy,B)] = B .
eq [xor(Dx,Dy,B)] = B .
eq [atom(A,B)] = B .
eq [previously(D,B)] = B .
eq [eventuallyPast(D,B)] = B .
eq [alwaysPast(D,B)] = B .
eq [since(Dx,Dy,B)] = B .
eq [interval(Dx,Dy,B)] = B .
eq [start(Dx,B)] = B .
eq [end(Dx,B)] = B .

```

The operation `mkDS` can be defined now. It basically follows the recursive semantics in Subsection 10.3.3, when the length of the trace is 1:

```

vars X Y : Formula .
op mkDS : Formula AtomState -> FormulaDS .
eq mkDS(true, As) = true .
eq mkDS(false, As) = false .
eq mkDS(A, As) = atom(A, (A{As} == true)) .
eq mkDS(X /\ Y, As) =
  and(mkDS(X,As), mkDS(Y,As),

```

```

    [mkDS(X,As)] and [mkDS(X,As)]) .
eq mkDS(X ++ Y, As) =
    xor(mkDS(X,As), mkDS(Y,As),
        [mkDS(X,As)] xor [mkDS(X,As)]) .
eq mkDS( (*)X, As) =
    previously(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( <*>X, As) =
    eventuallyPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS( [*]X, As) =
    alwaysPast(mkDS(X, As), [mkDS(X, As)]) .
eq mkDS(X Ss Y, As) =
    since(mkDS(X,As), mkDS(Y,As), [mkDS(Y,As)]) .
eq mkDS(X Sw Y, As) =
    since(mkDS(X,As), mkDS(Y,As),
        [mkDS(X,As)] or [mkDS(Y,As)]) .
eq mkDS(start(X), As) = start(mkDS(X,As),false) .
eq mkDS(end(X), As) = end(mkDS(X,As), false) .
eq mkDS([X,Y}s, As) =
    interval(mkDS(X,As), mkDS(Y,As),
        [mkDS(Y,As)] and not [mkDS(Y,As)]) .
eq mkDS([X,Y}w, As) =
    interval(mkDS(X,As), mkDS(Y,As),
        not [mkDS(Y,As)]) .

```

The data structure associated to a past time formula is essentially its syntax tree augmented with a boolean bit for each node. Each boolean bit will store the result of the satisfaction relation between the current execution trace and the corresponding subformula. The only thing left is to define how the formula data structures, or more precisely their bits, modify when a new event is received. This is defined below, using the operator `_{}_` : `FormulaDS AtomState -> FormulaDS` provided by the module `Formula`:

```

eq atom(A, B){As} = atom(A, (A{As} == true)) .
eq and(Dx, Dy, B){As} =
    and(Dx{As}, Dy{As}, [Dx{As}] and [Dy{As}]) .
eq xor(Dx, Dy, B){As} =
    xor(Dx{As}, Dy{As}, [Dx{As}] xor [Dy{As}]) .
eq previously(D,B){As} = previously(D{As}, [D]) .
eq eventuallyPast(D, B){As} =
    eventuallyPast(D{As}, [D{As}] or B) .
eq alwaysPast(D, B){As} =
    alwaysPast(D{As}, [D{As}] and B) .
eq since(Dx, Dy, B){As} =

```

```

      since(Dx{As}, Dy{As},
            [Dy{As}] or [Dx{As}] and B) .
eq start(Dx,B){As} =
  start(Dx{As}, [Dx{As}] and not B) .
eq end(Dx,B){As} =
  end(Dx{As}, not [Dx{As}] and B) .
eq interval(Dx, Dy, B){As} =
  interval(Dx{As}, Dy{As}, not [Dy{As}] and
            ([Dx{As}] or B)) .
endfm

```

The operator `_==_` is built-in and takes two terms of same sort, reduces them to their normal forms, and then returns true if they are equal and false otherwise.

10.4.5 Monitoring with Maude

In this subsection we give more details on how the actual rewriting based monitoring process works. When the JPAX system is started, the user is supposed to have already specified several formulae in a file containing monitoring requirements. The first thing JPAX does is to start a Maude process, load the past time LTL semantics described above, and then set Maude to run in its *loop mode*, which is an execution mode in which Maude maintains a state term which the user (potentially another process, such as JPAX) can modify interactively. Then JPAX sends Maude all the requirement formulae that the user wants to monitor. Maude stores them in its loop state and waits for JPAX to send events. Notice that the above is general and applies to any logic.

When JPAX receives the first event from the instrumented program that is relevant for the past time LTL analysis module, it just sends it to Maude. On receiving the first event, say `As`, Maude needs to generate the formula data structures for all the formulae to be monitored. It does so by replacing each formula `F` in the loop state by the normal form of the term `mkDS(F, As)`. Then it waits for JPAX to submit further events. Each time a new relevant event `As` is received by JPAX from the instrumented program, it just forwards it to Maude. Then Maude replaces each formula data structure `D` in its loop state by `D{As}` and then waits for further events. If at any moment `[D]` is false for the data structure `D` associated to a formula `F`, then Maude sends an error message to JPAX, which further warns the user appropriately.

It should be obvious that the runtime complexity of the rewriting monitoring algorithm is $O(m)$ to process an event, where m is the size of the past time LTL formula to monitor. That is, the algorithm only needs to traverse the data structure representing the formula bottom-up for each new event, and update one bit in each node. So the overall runtime complexity is $O(n \cdot m)$, where n is the number of events to be monitored. This is the best one can asymptotically hope from a runtime monitoring algorithm, but of course, there is room for even faster algorithms in practical situations, as the one presented in the next section. The main benefit of the rewriting algorithm presented in this section is that it falls under the general framework by which one can easily add or experiment with new monitoring logics within the JPAX system.

The Maude code performing the above steps is relatively straightforward but rather ugly, so we prefer not to present it here. Additionally, Maude's support for inter-process communication is planned to be changed soon, so this code would become soon obsolete.

10.5 Synthesizing Monitors for Safety Properties

The rewriting algorithm above is a very good choice in the context of the current version of JPAX, because it gives us flexibility and is efficient enough to process events at a faster rate than they can actually be sent by JPAX. However, there might be situations in which a full scale AC rewriting engine like Maude is not available, such as within an embedded system, or in which as little runtime overhead as possible is allowed, such as in real time applications. In this section we present a dynamic programming based algorithm, also based on the recursive semantics of past time LTL in Subsection 10.3.3, which takes as input a formula and generates source code which can further be compiled into an efficient executable monitor for that formula. This algorithm can be used in two different ways. On the one hand, it can be used as an efficient external monitor to take an action when a formula is violated, such as to report an error to a user, to reboot the system, to send a message, or even to generate a correcting task. On the other hand, it can be used in a context in which one allows past time LTL annotations in the source code of a program, where the logical annotations can be expanded into source code which is further compiled together with the original program. These two use modes, offline versus inline, are further explained in Subsection 10.5.4.

10.5.1 The Algorithm Illustrated by an Example

In this section we show via an example how to generate dynamic programming code for a concrete *ptLTL*-formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next subsection. Let $\uparrow p \rightarrow [q, \downarrow (r \vee s)]_s$ be the *ptLTL*-formula that we want to generate code for. The formula states: “whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s ”. The code translation depends on an enumeration of the subformulae of the formula that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, \dots, \varphi_8$ be such an enumeration:

$$\begin{aligned}\varphi_0 &= \uparrow p \rightarrow [q, \downarrow (r \vee s)]_s, \\ \varphi_1 &= \uparrow p, \\ \varphi_2 &= p, \\ \varphi_3 &= [q, \downarrow (r \vee s)]_s, \\ \varphi_4 &= q, \\ \varphi_5 &= \downarrow (r \vee s), \\ \varphi_6 &= r \vee s, \\ \varphi_7 &= r, \\ \varphi_8 &= s.\end{aligned}$$

Note that the formulae have here been enumerated in a post-order fashion. One could have chosen a breadth-first order, or any other enumeration, as long as the enumeration invariant is true.

The input to the generated program will be a finite trace $t = s_1 s_2 \dots s_n$ of n events. The generated program will maintain a state via a function $update : \mathbf{State} \times Event \rightarrow \mathbf{State}$, which updates the state with a given event.

In order to illustrate the dynamic programming aspect of the solution, one can imagine recursively defining a matrix $s[1..n, 0..8]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$. Then one can fill the table according to the recursive semantics of past time LTL as described in Subsection 10.3.3. This would be the standard way of regarding the above satisfaction problem as a dynamic programming problem. An important observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store the entire table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and $s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively.

It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

```

State  $state \leftarrow \{\}$ ;

bit  $pre[0..8]$ ;

bit  $now[0..8]$ ;

INPUT: trace  $t = s_1 s_2 \dots s_n$ ;

/* Initialization of  $state$  and  $pre$  */

 $state \leftarrow update(state, s_1)$ ;

 $pre[8] \leftarrow s(state)$ ;

 $pre[7] \leftarrow r(state)$ ;

 $pre[6] \leftarrow pre[7] \text{ or } pre[8]$ ;

 $pre[5] \leftarrow \mathbf{false}$ ;

 $pre[4] \leftarrow q(state)$ ;

 $pre[3] \leftarrow pre[4] \text{ and not } pre[5]$ ;

 $pre[2] \leftarrow p(state)$ ;

 $pre[1] \leftarrow \mathbf{false}$ ;

 $pre[0] \leftarrow \text{not } pre[1] \text{ or } pre[3]$ ;

/* Event interpretation loop */

for  $i = 2$  to  $n$  do {

     $state \leftarrow update(state, s_i)$ ;

     $now[8] \leftarrow s(state)$ ;

     $now[7] \leftarrow r(state)$ ;

     $now[6] \leftarrow now[7] \text{ or } now[8]$ ;

     $now[5] \leftarrow \text{not } now[6] \text{ and } pre[6]$ ;

```

```

    now[4] ← q(state);

    now[3] ← (pre[3] or now[4]) and not now[5];

    now[2] ← p(state);

    now[1] ← now[2] and not pre[2];

    now[0] ← not now[1] or now[3];

    if now[0] = 0 then

        output(■property violated■);

    pre ← now;

};

```

In the following we explain the generated program.

Declarations Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state.

Initialization The initialization phase consists of initializing the *state* variable and the *pre* array. The first event s_1 of the event list is used to initialize the *state* variable. The *pre* array is initialized by evaluating all subformulae bottom up, starting with highest formula numbers, and assigning these values to the corresponding elements of the *pre* array; hence, for any $i \in \{0 \dots 8\}$ $pre[i]$ is assigned the initial value of formula φ_i . The *pre* array is initialized in such a way as to maintain the view that the initial state is supposed stationary before monitoring is started. This in particular means that $\uparrow p$ is false, as well as is $\downarrow (r \vee s)$, since there is no change in state (indices 1 and 5). The interval operator has the obvious initial interpretation: the first argument must be true and the second false for the formula to be true (index 3). Propositions are true if they hold in the initial state (indices 2, 4, 7 and 8), and boolean operators are interpreted the standard way (indices 0, 6).

Event Loop The main evaluation loop goes through the event trace, starting from the second event. For each such event, the state is updated, followed by assignments to the *now* array in a bottom-up fashion similar to the initialization of the *pre* array: the array elements are assigned values from higher index values to lower index values, corresponding to the values of the corresponding subformulae. Propositional boolean operators are interpreted the standard way (indices 0 and 6). The formula $\uparrow p$ is true if p is true now and not true in the previous state (index 1). Similarly with the formula $\downarrow (r \vee s)$ (index 5). The formula $[q, \downarrow (r \vee s)]_s$ is true if either the formula was true in the previous state, or q is true in the current state, and in addition $\downarrow (r \vee s)$ is not true in the current state (index 3). At the end of the loop an error message is issued if *now*[0], the value of the whole formula, has the value 0 in the current state. Finally, the entire *now* array is copied into *pre*.

Given a fixed *ptLTL* formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the *ptLTL* formula. The memory required is constant, since the length of the two arrays is the size of the *ptLTL* formula. However, one may want to also include the size of the formula, say m , into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while memory required is $2 \cdot (m + 1)$ bits. The authors conjecture that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations are possible (see Section 10.5.3).

10.5.2 The Algorithm Formalized

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from a *ptLTL*-formula. It takes as input a formula and generates a program as the one above, containing a “for” loop which traverses the trace of events, while validating or invalidating the formula. The generated program is printed using the function **output**, which takes one or more string or integer parameters which are concatenated in the output. This algorithm is designed to generate pseudocode, but it can easily be adapted to generate code in any imperative programming language:

INPUT: past time LTL formula φ

let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the subformulae of φ ;

```

output("State  $state \leftarrow \{\}$ ");
output("bit  $pre[0..m]$ ");
output("bit  $now[0..m]$ ");
output("INPUT: trace  $t = s_1 s_2 \dots s_n$ ");
output("/* Initialization of  $state$  and  $pre$  */");
output("state  $\leftarrow update(state, s_1)$ ");
for  $j = m$  downto 0 do {
    output("     $pre[$ ,  $j$ ,  $]$   $\leftarrow$  ");
    if  $\varphi_j$  is a variable then
        output( $\varphi_j$ , "(state)");
    if  $\varphi_j$  is true then output("true");
    if  $\varphi_j$  is false then output("false");
    if  $\varphi_j = \neg \varphi_{j'}$  then
        output("not  $pre[$ ,  $j'$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} \text{ op } \varphi_{j_2}$  then
        output("pre[ $, j_1$ ,  $]$  op pre[ $, j_2$ ,  $]$ ");
    if  $\varphi_j = \odot \varphi_{j_1}$  then
        output("pre[ $, j_1$ ,  $]$ ");
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output("pre[ $, j_1$ ,  $]$ ");
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output("pre[ $, j_1$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
        output("pre[ $, j_2$ ,  $]$ ");

```

```

if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
    output("pre[" ,  $j_1$ , "]" or pre[" ,  $j_2$ , "]" );
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
    output("pre[" ,  $j_1$ , "]" and not pre[" ,  $j_2$ , "]" );
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
    output("not pre[" ,  $j_2$ , "]" );
if  $\varphi_j = \uparrow \varphi_{j'}$  then output("false;");
if  $\varphi_j = \downarrow \varphi_{j'}$  then output("false;");
};
output("/ * Event interpretation loop * /");
output("for  $i = 2$  to  $n$  do {");
for  $j = m$  downto 0 do {
    output("    now[" ,  $j$ , "]"  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output( $\varphi_j$ , "(state);");
    if  $\varphi_j$  is true then output("true;");
    if  $\varphi_j$  is false then output("false;");
    if  $\varphi_j = \neg \varphi_{j'}$  then output("not now[" ,  $j'$ , "]" );
    if  $\varphi_j = \varphi_{j_1} op \varphi_{j_2}$  then
        output("now[" ,  $j_1$ , "]" op now[" ,  $j_2$ , "]" );
    if  $\varphi_j = \odot \varphi_{j_1}$  then output("pre[" ,  $j_1$ , "]" );
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output("pre[" ,  $j$ , "]" or now[" ,  $j_1$ , "]" );
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output("pre[" ,  $j$ , "]" and now[" ,  $j_1$ , "]" );

```

```

if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
  output("(pre", j, "[" and now", j1, "[") or
    now", j2, "];");
if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
  output("(pre", j, "[" and now", j1, "[") or
    now", j2, "];");
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
  output("(pre", j, "[" or now", j1, "[") and
    not now", j2, "];");
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
  output("(pre", j, "[" or now", j1, "[") and
    not now", j2, "];");
if  $\varphi_j = \uparrow \varphi_{j'}$  then
  output("(now", j', "[" and
    not pre", j', "];");
if  $\varphi_j = \downarrow \varphi_{j'}$  then
  output("(not now", j', "[" and
    pre", j', "];");
};
output("    if now[0] = 0 then
  output("■property violated■");
output("    pre ← now;");
output("}");

```

op is any binary propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of this algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the list of all

subformulae. Because of the recursive nature of *ptLTL*, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

Before we generate the main loop, we should first generate code for initializing the array *pre*[0..*m*], basically giving it the truth values of the subformulae on the initial state, conceptually being an infinite trace with repeated occurrences of the initial state. After that, the generated main event loop will process the events. The loop body will update/calculate the array *now* and in the end will move it into the array *pre* to serve as basis for the next iteration. After each iteration *i*, *now*[0] tells whether the formula is validated by the trace $s_1s_2\dots s_i$.

Since the formula enumeration procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an *ptLTL* formula in linear time with the size of the formula. The boolean operations used above are usually very efficiently implemented on any microprocessor and the arrays of bits *pre* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast. We shall next illustrate how such optimizations can be part of the translation algorithm.

10.5.3 Optimizing the Generated Code

The generated code presented in Subsection 10.5.1 is not optimal. Even though a smart compiler can in principle generate good machine code from it, it is still worth exploring ways to synthesize directly optimized code especially because there are some attributes that are specific to the runtime observer which a compiler cannot take into consideration.

A first observation is that not all the bits in *pre* are needed, but only those which are used at the next iteration, namely 2, 3, and 6. Therefore, only a bit per temporal operator is needed, thereby reducing significantly the memory required by the generated algorithm. Then the body of the generated “for” loop becomes after (blind) substitution (we don’t consider the initialization code here):

$$\begin{aligned} state &\leftarrow update(state, s_i) \\ now[3] &\leftarrow r(state) \text{ or } s(state) \end{aligned}$$

```

now[2] ← (pre[2] or q(state)) and
          not (not now[3] and pre[3])
now[1] ← p(state)
if ((not (now[1] and not pre[1]) or now[2]) = 0)
  then output(property violated);

```

that can be further optimized by boolean simplifications:

```

state ← update(state, si)
now[3] ← r(state) or s(state)
now[2] ← (pre[2] or q(state)) and
          (now[3] or not pre[3])
now[1] ← p(state)
if (now[1] and not pre[1] and not now[2])
  then output(property violated);

```

The most expensive part of the code above is the function calls, namely $p(state)$, $q(state)$, $r(state)$, and $s(state)$. Depending upon the runtime requirements, the execution time of these functions may vary significantly. However, since one of the major concerns of monitoring is to affect the normal execution of the monitored program as little as possible, especially in the inline monitoring approach, one would of course want to evaluate the atomic predicates on states only if really needed, or rather to evaluate only those that, probabilistically, add a minimum cost. Since we don't want to count on an optimizing compiler, we prefer to store the boolean formula as some kind of binary decision diagram, more precisely, as a term over the conditional operation $_{?} : _$, where ' $e_1 ? e_2 : e_3$ ' means: "if e_1 then e_2 else e_3 ". For example, $pre[3] ? pre[2] ? now[3] : q(state) : pre[2] ? 1 : q(state)$ (see [126] for a formal definition). Therefore, one is faced with the following optimization problem:

Given a boolean formula φ using propositions a_1, a_2, \dots, a_n of costs c_1, c_2, \dots, c_n , respectively, find a $(_{?} : _)$ -expression that optimally implements φ .

We have implemented a procedure in Maude, on top of propositional calculus, which generates all correct $(_?_ : _)$ -expressions for φ , admittedly a potentially exponential number in the number of distinct atomic propositions in φ , and then chooses the shortest in size, ignoring the costs. Applied on the code above, it yields:

```

state ← update(state, si)

now[3] ← r(state) ? 1 : s(state)

now[2] ← pre[3] ? pre[2] ? now[3] :

      q(state) : pre[2] ? 1 : q(state)

now[1] ← p(state)

if (pre[1] ? 0 : now[2] ? 0 : now[1])

  then output(■property violated■);

```

We would like to extend our procedure to take the evaluation costs of predicates into consideration. These costs can either be provided by the user of the system or be calculated automatically by a static analysis of predicates' code, or even be estimated by executing the predicates on a sample of states. However, based on our examples so far, we conjecture at this incipient stage that, given a boolean formula φ in which all the atomic propositions have the same cost, the probabilistically runtime optimal $(_?_ : _)$ -expression implementing φ is *exactly* the one which is smallest in size.

A further optimization would be to generate directly machine code instead of using a compiler. Then the arrays of bits *now* and *pre* can be stored in two registers, which would be all the memory needed. Since all the operations executed are bit operations, the generated code is expected to be very fast. One could even imagine hardware implementations of past time monitors, using the same ideas, in order to enforce safety requirements on physical devices.

10.5.4 Implementation of Offline and Inline Monitoring

In this section we briefly describe our efforts to implement the above described algorithm to create monitors for observing the execution of Java programs in PathExplorer. We present two approaches that we have pursued. In the *off-line* approach we create a monitor that runs in parallel with the

executing program, potentially on a different computer, receiving events from the running program, and checking on-the-fly that the formulae are satisfied. In this approach the formulae to be checked are given in a separate specification. In the *inline* approach, formulae are written as comments in the program text, and are then expanded into Java code that is inserted after the comments.

Offline Monitoring

The code generator for off-line monitoring has been written in Java, using JavaCC [159], an environment for writing parsers and for generating and manipulating abstract syntax trees. The input to the code generator is a specification given in a file separate from the program. The specification for our example looks as follows (the default interpretation of intervals is “strong”):

```
specification Example is
P = start(p) -> [q,end(r|s));
end
```

Several named formulae can be listed; here we have only included one, named P. The translator reads this specification and generates a single Java class, called **Formulae**, which contains all the machinery for evaluating all the formulae (in this case one) in the specification. This class must then be compiled and instantiated as part of the monitor. The class contains an **evaluate()** method which is applied after each state change and which will evaluate all the formulae. The class constructor takes as parameter a reference to the object that represents the state, such that any updates to the state by the monitor, based on received events, can be seen by the **evaluate()** method. The generated **Formulae** class for the above specification looks as follows:

```
class Formulae{
  abstract class Formula{
    protected String name;  protected State state;
    protected boolean[] pre; protected boolean[] now;

    public Formula(String name,State state){
      this.name = name; this.state = state;
    }
    public String getName(){return name;}
    public abstract boolean evaluate();
  }
  private List formulae = new ArrayList();
  public void evaluate(){
```

```

        Iterator it = formulae.iterator();
        while(it.hasNext()){
            Formula formula = (Formula)it.next();
            if(!formula.evaluate()){
                System.out.println("Property " + formula.getName() +
                                   " violated");
            }
        }
    }
    class Formula_P extends Formula{
        public boolean evaluate(){
            now[8] = state.holds("s");
            now[7] = state.holds("r");
            now[6] = now[7] || now[8];
            now[5] = !now[6] && pre[6];
            now[4] = state.holds("q");
            now[3] = (pre[3] || now[4]) && !now[5];
            now[2] = state.holds("p");
            now[1] = now[2] && !pre[2];
            now[0] = !now[1] || now[3];
            System.arraycopy(now,0,pre,0,9);
            return now[0];
        }
        public Formula_P(State state){
            super("P",state);
            pre = new boolean[9]; now = new boolean[9];
            pre[8] = state.holds("s");
            pre[7] = state.holds("r");
            pre[6] = pre[7] || pre[8];
            pre[5] = false;
            pre[4] = state.holds("q");
            pre[3] = pre[4] && !pre[5];
            pre[2] = state.holds("p");
            pre[1] = false;
            pre[0] = !pre[1] || pre[3];
        }
    }
    public Formulae(State state){
        formulae.add(new Formula_P(state));
    }
}

```

The class contains an inner abstract³ class **Formula** and, in the general case, an inner class **Formula_X** extending the **Formula** class for each formula in the specification, where **X** is the formula's name. In our case there is one such **Formula_P** class. The abstract **Formula** class declares the **pre** and **now** arrays, without giving them any size, since this is formula specific. An abstract **evaluate** method is also declared. The class **Formula_P** contains the real definition of this **evaluate()** method. The constructor for this class in addition initializes the sizes of **pre** and **now** depending on the size of the

³An abstract class is a class where some methods are abstract, by having no body. Implementations for these methods will be provided in extending subclasses.

formula, and also initializes the `pre` array.

In order to handle the general case where several formulae occur in the specification, and hence many `Formula_X` classes are defined, we need to create instances for all these classes and store them in some data structure where they can be accessed by the outermost `evaluate()` method. The `formulae` list variable is initialized to contain all these instances when the constructor of the `Formulae` class is called. The outermost `evaluate()` method, on each invocation, goes through this list and calls `evaluate()` on each single formula object.

Inline Monitoring

The general architecture of PAX was mainly designed for offline monitoring in order to accommodate applications where the source code is not available or where the monitored process is not even a program, but some kind of physical device. However, it is often the case that the source code of an application *is* available and that one is willing to accept extra code for testing purposes. Inline monitoring has actually higher precision because one knows exactly where an event was emitted in the execution of the program. Moreover, one can even throw exceptions when a safety property is violated, like in Temporal Rover [87], so the running program has the possibility to recover from an erroneous execution or to guide its execution in order to avoid undesired behaviors.

In order to provide support for inline monitoring, we developed some simple scripts that replace temporal annotations in Java source code by actual monitoring code, which throws an exception when the formula is violated. In [132] we show an example of expanded code for future time LTL. The “for” loop and the update of the state in the generic algorithm in Section 10.5.1 are not needed anymore because the atomic predicates use directly the current state of the program when the expanded code is reached during the execution. In [53] the tool Java-MoP is described, which implements the presented algorithm as a logic plug-in for inline monitoring (as well as for off-line monitoring).

The following code snippets illustrate the inline approach. Assume a class `A`, that defines four integer variables and a method `m`, which contains the past time temporal logic formula from above. Now the propositions `p`, `q`, `r` and `s` are defined to refer to the four variables. The intention is that whenever the program point of the comment is reached, the formula will be evaluated.

```

class A{
  int a,b,c,d;
  void m(){
    ...
    /* @monitor
       proposition p = a>0;
       proposition q = b>0;
       proposition r = c>0;
       proposition s = d>0;
       property P = start(p) -> [q,end(r|s));
    */
    ...
  }
}

```

This class is now automatically translated into the following, where code representing the semantics of the formula has been inserted at the position of the formula comment, and in the constructor:

```

class A{
  int a,b,c,d;
  boolean[] pre = new boolean[9];
  boolean[] now = new boolean[9];

  public A(){
    pre[8] = d>0;
    pre[7] = c>0;
    pre[6] = pre[7] || pre[8];
    pre[5] = false;
    pre[4] = b>0;
    pre[3] = pre[4] && !pre[5];
    pre[2] = a>0;
    pre[1] = false;
    pre[0] = !pre[1] || pre[3];
  }

  void m(){
    ...
    now[8] = d>0;
    now[7] = c>0;
    now[6] = now[7] || now[8];
    now[5] = !now[6] && pre[6];
    now[4] = b>0;
    now[3] = (pre[3] || now[4]) && !now[5];
    now[2] = a>0;
    now[1] = now[2] && !pre[2];
    now[0] = !now[1] || now[3];
    System.arraycopy(now,0,pre,0,9);
    if(!now[0])throw Violated("P");
    ...
  }
}

```

It is essentially the same code as in the offline case, except that the looping constructs have been removed.

It is inline monitoring that motivated us to optimize the generated code as much as possible as in Subsection 10.5.3. Since the running program and the monitor are a single process now, the time needed to execute the monitoring code can significantly influence the otherwise normal execution of the monitored program.

10.6 Conclusion

Two efficient algorithms for monitoring safety requirements expressed using past time linear temporal logic were presented, one based on rewriting and implemented in Maude, and the other based on dynamic programming, synthesizing specialized monitors from formulae. They both check that a finite sequence of events emitted by a running program satisfies a formula. Operators convenient for monitoring were considered and shown equivalent to standard past time temporal operators.

These algorithms have been implemented in PathExplorer, a runtime verification tool currently under development. The synthesis algorithm has also been implemented (as a plug-in) in the Java-MoP tool [53], which is a general framework for supporting program monitoring for user provided logics; and in the JMPaX tool [240], which extends part of this work to partial order models instead of simple traces.

It is our intention to investigate how the presented algorithms can be refined to work for a logic that combines past and future time temporal logic and that can refer to real-time and data values. Other kinds of runtime verification are also investigated, such as, for example, techniques for detecting error potentials in multi-threaded programs. Recent work on detecting high-level data races is described in [17].

A number of experiments have been carried out with PathExplorer on a planetary rover application written in 35,000 lines of C++. The experiments range from concurrency analysis (deadlock and data race analysis) to monitoring of temporal logic formulae combined with test case generation, as described in [18]. A model checker is used to generate test cases, where a test case consists of input to the application plus a set of temporal formulae that the execution of the application on that input must satisfy. When running this testing environment, hundreds of test cases are generated and the execution of these are monitored against the generated formulae. Initial

experiments have been made with a logic that combines past time and future time temporal logic and supports real-time and data reasoning. A bug was detected in the rover application in the very first such experiment we made. A thread did not detect a premature termination of a certain task in a timely manner. The programmer had forgotten to insert this termination check and was reminded by a single run of the testing environment. This testing environment is planned to become part of the rover application programmer's testing toolbox.

new stuff below

10.7 Optimal Monitoring of “Always Past” Temporal Safety

A monitor synthesis algorithm from linear temporal logic (LTL) safety formulae of the form $\Box\varphi$ where φ is a past time LTL formula was presented in [129]. The generated monitors implemented the recursive semantics of past-time LTL using a dynamic programming technique, and needed $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Some compiler-like optimizations of the generated monitors were also proposed in [129], which would further reduce the required space. It is not clear how much the required space could be reduced by applying those optimizations.

We here show how to generate using a divide-and-conquer technique directly monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ .

10.7.1 The Monitor Synthesis Algorithm

For simplicity, we assume only two past operators, namely \circ (previously) and \mathcal{S} (since). Let us first note that one cannot asymptotically reduce the space requirements below $\Omega(k)$, where k is the number of temporal operators appearing in the formula to monitor φ . Indeed, one can take $\varphi = (\#_1 \rightarrow t_1) \wedge \cdots \wedge (\#_k \rightarrow t_k)$, where for each $1 \leq i \leq k$, $\#_i$ is some event and t_i is some temporal formula containing precisely one past temporal operator, i.e., a \circ or a \mathcal{S} . Any monitor for φ must directly or indirectly store the status of each t_i at every event, to be able to react accordingly in case the next event is some $\#_i$. Assuming that the events $\#_i$ are distinct

and that the formulae t_i are unrelated, then the monitor needs to distinguish among 2^k possible states, so it needs $\Omega(k)$ space.

In what follows, we assume the usual recursive semantics of LTL, also presented below, restricted to safety formulae of the form $\Box\varphi$, where φ is a past-time LTL. We adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [129], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring.

Definition 45 (adapted from [187]) *LTL formulae of the form $\Box\varphi$ (read “always φ ”), where φ is a past-time LTL formula, are called LTL safety formulae; we may call them just safety formulae when LTL is understood from the context. An infinite trace $u \in \Sigma^\omega$ satisfies $\Box\varphi$, written $u \models \Box\varphi$, iff each $w \in \text{prefixes}(u)$ satisfies the past-time LTL formula φ , written also $w \models \varphi$ and defined inductively as follows:*

$w \models \text{true}$	<i>is always true,</i>
$ws \models a$	<i>iff $a(s)$ holds,</i>
$w \models \neg\varphi$	<i>iff $w \not\models \varphi$,</i>
$w \models \varphi_1 \wedge \varphi_2$	<i>iff $w \models \varphi_1$ and $w \models \varphi_2$,</i>
$ws \models \circ\varphi$	<i>iff $w \models \varphi$,</i>
$ws \models \varphi_1 \mathcal{S} \varphi_2$	<i>iff $ws \models \varphi_2$ or $ws \models F\varphi$ and $w \models \varphi_1 \mathcal{S} \varphi_2$</i>
$\epsilon \models \varphi$	<i>is false otherwise</i>

Given safety formula $\Box\varphi$, we let $\mathcal{L}(\Box\varphi) \subseteq \Sigma^\omega$ be the set $\{u \in \Sigma^\omega \mid u \models \Box\varphi\}$.

Proposition 16 $\mathcal{L}(\Box\varphi) \in \text{Safety}^\omega$ for any past-time LTL formula φ .

Proof: By the definition of $\mathcal{L}(\Box\varphi)$ in Definition 45 and the definition of $\Box P$ in Definition 12, one can easily note that $\mathcal{L}(\Box\varphi) = \Box\mathcal{L}(\varphi)$, where $\mathcal{L}(\varphi) = \{w \in \Sigma^* \mid w \models \varphi\}$. Therefore, $\mathcal{L}(\Box\varphi) \in \text{Safety}_\Box^\omega$. The rest follows by Theorem 5. \square

Let us next investigate the problem of monitoring safety properties $P \in \text{Safety}^\omega$ expressed as languages of safety formulae, that is, $P = \mathcal{L}(\Box\varphi)$ for some past-time LTL formula φ . Because of the recursive nature of the satisfaction relation, a first important observation is that the generated monitor only needs to store information regarding the status of temporal operators from the previous state. More precisely, the monitor needs one bit per temporal operator, keeping the satisfaction status of the subformula corresponding to that temporal operator; when a new state is received,

the satisfaction status of the subformula is recalculated according to the recursive semantics above and then the bit is updated. The order in which the temporal operators are processed when a new state is received is important: the nested operators must be processed first.

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as we did in [129], because different applications may choose different implementation paradigms. For example, we are currently using rewriting techniques to implement the monitor synthesis algorithms in MOP [56]; Section 10.7.2 shows our complete Maude rewriting implementation of the subsequent monitor synthesis algorithm.

Step 1 Let $\varphi_1, \dots, \varphi_k$ be the k subformulae of φ corresponding to temporal operators, such that, if φ_i is a subformula of φ_j , then $i < j$; this can be easily achieved by a DFS traversal of φ .

Step 2 Let $bit[1..k]$ be a vector of k bits initialized to 0 (or false); $bit[i]$ will store information related to φ_i from the previous state:

- if $\varphi_i = \odot\psi$ then $bit[i]$ says if ψ was satisfied at the previous state;
- if $\varphi_i = \psi \mathcal{S} \psi'$ then $bit[i]$ says if φ_i was satisfied at the previous step.

Step 3 Let $bit'[1..k]$ be another vector of k bits; this will be used to store temporary results, which will be moved eventually into the vector $bit[1..k]$.

Step 4 Generate a loop that executes whenever a new state s is available; the body of the loop executes the following code:

Step 4.1 For each i from 1 to k execute a bit assignment as follows, where for a subformula ψ of φ , $\overline{\psi}$ is the boolean expression replacing in ψ each non-nested temporal subformula φ_j by $bit[j]$ if φ_j is a “previously” formula or by $bit'[j]$ if φ_j is a “since” formula, and each remaining atomic proposition a by its satisfaction in the current state, $a(s)$:

- if $\varphi_i = \odot\psi$ then generate the assignment $bit'[i] := \overline{\psi}$
- if $\varphi_i = \psi \mathcal{S} \psi'$ then generate the assignment $bit'[i] := \overline{\psi'} \vee \overline{\psi} \wedge bit[i]$

Step 4.2 Generate the conditional: if $\bar{\varphi}$ is false then error (formula violated)

Step 4.3 Generate code to move the contents of $bit'[1..k]$ into $bit[1..k]$.

Note that the generated monitors are well-defined, because each time a $\bar{\psi}$ boolean expression is generated, all the bits in $bit'[1..k]$ that are needed are already calculated. One can also perform boolean simplifications when calculating $\bar{\psi}$ to reduce runtime overhead even further. For example, in our implementation that also generated the code below (see Section 10.7.2), we used the simplification $\neg\neg\psi = \psi$. To illustrate the monitor generation algorithm above, let us consider the past time LTL formula: $\varphi = \neg(a \wedge \neg(\circ b \wedge (c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))))$. Step 1 produces the following enumeration of φ 's subformulae: $\varphi_1 = \circ b$, $\varphi_2 = \neg e \mathcal{S} f$, and $\varphi_3 = c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))$. The other steps eventually generate the code:

```

bit[1..3] := false;      // three global bits
foreach new state s do {
    // first update the bits in a consistent order
    bit'[1] := b(s);
    bit'[2] := f(s)  $\vee$  ( $\neg e(s) \wedge bit[2]$ );
    bit'[3] := d(s)  $\wedge$  bit'[2]  $\vee$  (c(s)  $\wedge$  bit[3]);
    // then check whether the formula is violated
    if a(s)  $\wedge$   $\neg(bit[1] \wedge bit'[3])$  then Error;
    // finally, update the state of the monitor
    bit[1..3] := bit'[1..3]
}

```

It is easy to see that for any past LTL formula φ of k temporal operators, the state of the generated monitor is encoded on k bits, namely the vector $bit[1..k]$. The runtime of the generated monitor is still $O(|\varphi|)$, because each temporal operator in φ results in an assignment and a read operation in the monitor, while each boolean operator in φ is “executed” by the monitor.

10.7.2 A Maude Implementation of the Monitor Synthesizer

We here show a term rewriting implementation of the algorithm above, using the Maude system [73]. Implementations in other languages are obviously also possible; however, rewriting proved to be an elegant means to generate monitors from logical formulae in several other contexts, and so seems to be here. In what follows we show the complete Maude code that takes as input

a formula, parses it, generates the monitor, and then pretty prints it. We use the K technique here [225], which is a rewriting-based language and/or logic definitional technique; to use K, one needs to first upload the generic, i.e., application-independent, module discussed at the end of this section.

Atomic Predicates

We start by defining the atomic state predicates that one can use in formulae. These can be either identifiers (of the form 'a', 'abc', 'a123, etc.; these are provided by the Maude builtin module QID):

```
fmod PREDICATE is
  --- atomic predicates can be quoted identifiers
  protecting QID .
  sort Predicate .
  subsort Qid < Predicate .
endfm
```

Syntax of Formulae

Let us next define the syntax of formulae. We here use Maude's mixfix notation for defining syntax as algebraic operators, where underscores stay for arguments. Also, note that operators are assigned precedences (declared as operator attributes), to relive the user from writing parentheses (the lower the precedence the tighter the binding):

```
fmod SYNTAX is
  protecting PREDICATE .
  sort Formula .
  subsort Predicate < Formula .
  op !_ : Formula -> Formula [prec 20] .
  op _/\_ : Formula Formula -> Formula [prec 23] .
  op 0_ : Formula -> Formula [prec 21] .
  op _S_ : Formula Formula -> Formula [prec 22] .
endfm
```

Target Language

We are done with the input language. Let us now define the output language. We need a very simple language for implementing the generated monitors, namely one with limited assignment, conditional and looping. The generated code, as well as the target language, play no role in this paper; one is expected to change the language below to one's desired target language (Java, C,

C#, assembler, etc.). Our chosen language below has bits, expressions, statements and code. Bits are also expressions; code is a list of statements composed sequentially using “;” or just concatenation. The syntax below is also making use of precedence attributes. The `format` attributes are useful solely for pretty-printing reasons (see Maude’s manual [73] for details on formatting):

```
fmod CODE is
  --- syntax for the generated code
  protecting PREDICATE + INT + STRING .
  sorts Bit Exp Statement Code .
  subsorts Bit < Exp .
  subsort Statement < Code .
  ops (bit[_]) (bit'[_]) : Nat -> Bit .
  ops (bit[1 .. _]) (bit'[1 .. _]) : Int -> Bit .
  op _(s) : Predicate -> Exp [prec 0] .
  ops true false : -> Exp .
  op !_ : Exp -> Exp [prec 20] .
  op _/\_ : Exp Exp -> Exp [prec 23] .
  op _\/_ : Exp Exp -> Exp [prec 24] .
  op _:=_ : Exp Exp -> Statement [prec 27 format(ni d d d)] .
  op if_then_ : Exp Statement -> Statement
    [format(ni d d ++ --) prec 30] .
  op foreach new state s do _ : Code -> Statement
    [format(n d d d d s++ --n)] .
  op Error : -> Statement [format(ni d)] .
  op //_ : String -> Statement [format(ni d d)] .
  op nil : -> Code .
  op _;_ : Code Code -> Code [assoc id: nil prec 40] .
  op __ : Code Code -> Code [assoc id: nil prec 40] .
  op {_} : Code -> Statement [format(d d --ni ++)] .
  --- code simplification rules
  var B : Exp .
  eq !! B = B .
endfm
```

The following module defines the actual monitor synthesis algorithm. We use the K definitional technique here, because it yields a very compact implementation. K is centered on the basic intuition of *computation*; computations are encoded as first-order data-structures that “evolve”, via rewriting, to *results*. Computations are sequentialized using the list constructor “_>_”; thus, if K and K’ are computations, then K -> K’ is the computation consisting of K followed by K’. Computations may eventually yield results; for example, K -> K’ may rewrite (in context) to R -> K’, meaning that R is the result that K reduces to. An important feature of K is that one can

schedule lists of tasks for reduction; for example, $[K1, K2, K3] \rightarrow K$ may eventually reduce to $[R1, R2, R3] \rightarrow K$, where $R1$, $R2$, and $R3$ are the results that $K1$, $K2$, and $K3$ reduce to, in this order. To use K , one needs to import the module K discussed at the end of this section. The equations of the module K (three in total) are all about reducing a list of computations to a list of results, supposing that one knows how to reduce one computation to one result.

K is a definitional framework that is generic in computations and results. More precisely, it provides sorts `KComputation` and `KResult`, and expects its user to define the desired computations and results, as well as rules to reduce a computation to a result. Computations typically can be reduced to results only in context; to facilitate this, K provides a sort `KConfiguration`, which is also supposed to be populated accordingly. The sort `KConfiguration` is a multi-set sort over a sort `KConfigurationItem`, where the multi-set constructor is just concatenation; also, the sort `KComputation` is a list sort over `KComputationItem`, where the list constructor is `_->_`. To make use of K , one needs to first define constructors for the sorts `KConfigurationItem`, `KComputationItem` and `KResult`, and then to define how each computation item reduces to a result.

In our case, the computations are the formulae or subformulae that still need to be processed, and the results are the corresponding boolean expressions that need to be checked in the current (generated code) context to see whether the formula has been violated or not. We define the following additional constructors: we add four constructors for configurations, namely “`k`” that wraps the current computation, “`code`” that wraps the current generated code, and “`nextBit`” that wraps the next available bit; we add one main constructor for computations, “`form`”, that wraps a formula, and one constant computation item per operator in the input language (the later is needed to know how to combine back the results of the corresponding subexpressions; finally, we add one constructor for results, “`exp`”, that wraps a boolean expression.

The formula is processed in a depth-first-order, following a divide-and-conquer philosophy. Each subformula is decomposed into a list of computation subtasks consisting of its subformulae, then the corresponding results are composed back into a result corresponding to the original subformula. Recall that equations/rules apply wherever they match, not only at the top. Let us only discuss the two equations defining the “since” (`_S_`), the last two in the module below. The first one is straightforward: it decomposes

the task of processing $F1 \ S \ F2$ to the subtasks of processing $F1$ and $F2$; the computation item S is placed in the computation structure to prepare the terrain for the next equation. The next equation applies after $F1$ and $F2$ have been processed, say to expressions $B1$ and $B2$, respectively; if C is the code generated so far and if $I+1$ is the next bit available, then the boolean expression corresponding to the current since formula is indeed $\text{bit}'(I+1)$, provided that one adds the corresponding code capturing the recursive semantics of since to the generated code.

```
fmod MONITOR-GENERATION is
  protecting K + SYNTAX + CODE .
  op k : KComputation -> KConfigurationItem .
  op code : Code -> KConfigurationItem .
  op nextBit : Nat -> KConfigurationItem .
  op process : Formula -> KConfigurationItem .
  op form : Formula -> KComputationItem .
  op exp : Exp -> KResult .
  ops ! /\ 0 S : -> KComputationItem .
  var P : Predicate . vars F F1 F2 : Formula . var C : Code .
  var I : Nat . vars B B1 B2 : Exp . var K : KComputation .
  eq process(F) = k(form(F)) code(nil) nextBit(0) .
  eq k(form(P) -> K) = k(exp(P(s)) -> K) .
  eq form(! F) = form(F) -> ! .
  eq exp(B) -> ! = exp(! B) .
  eq form(F1 /\ F2) = [form(F1),form(F2)] -> /\ .
  eq [exp(B1),exp(B2)] -> /\ = exp(B1 /\ B2) .
  eq form(0 F) = form(F) -> 0 .
  eq k(exp(B) -> 0 -> K) code(C) nextBit(I)
    = k(exp(bit[I + 1]) -> K) code(C ; bit'[I + 1] := B)
      nextBit(I + 1) .
  eq form(F1 S F2) = [form(F1), form(F2)] -> S .
  eq k([exp(B1),exp(B2)] -> S -> K) code(C) nextBit(I)
    = k(exp(bit'[I + 1]) -> K)
      code(C ; bit'[I + 1] := B2 \/ B1 /\ bit[I + 1]) nextBit(I + 1) .
endfm
```

Putting It All together

The following module plugs the code generated above into the general pattern:

```
fmod PRETTY-PRINT is
  protecting MONITOR-GENERATION .
  sort Monitor .
  op genMonitor : Formula -> Code .
  op makeMonitor : KConfiguration -> Code .
```

```

var F : Formula . var B : Exp . var C : Code . vars N M : Nat .
eq genMonitor(F) = makeMonitor(process(F)) .
eq makeMonitor(k(exp(B)) code(C) nextBit(N))
= bit[1 .. N] := false ;
  foreach new state s do {
    // "first update the bits in a consistent order"
    C ;
    // "then check whether the formula is violated"
    if !(B) then Error ;
    // "finally, update the state of the monitor"
    bit[1 .. N] := bit'[1 .. N]
  } .
endfm

```

Our implementation of the monitor synthesizer is now complete. To use it, one can ask Maude reduce terms of the form `genMonitor(F)`, where `F` is the formula that one wants to generate into a monitor. For example:

```

reduce genMonitor(
  !('a /\ !(0 'b /\ 'c S ('d /\ (! 'e S 'f))))
) .

```

For the formula above, Maude will give the expected answer, pretty printed as follows:

```

\|||||/
--- Welcome to Maude ---
/|||||
Maude 2.2 built: Mar 15 2006 16:37:22
Copyright 1997-2005 SRI International
Sat Jan 27 12:01:20 2007
Maude> in p
=====
fmod K
=====
fmod PREDICATE
=====
fmod SYNTAX
=====
fmod CODE
=====
fmod MONITOR-GENERATION
=====
fmod PRETTY-PRINT
=====

```

```

reduce in PRETTY-PRINT :
  genMonitor(! ('a /\ ! (0 'b /\ 'c S ('d /\ ! 'e S 'f)))) .
rewrites: 46 in -93406740ms cpu (1ms real) (~ rewrites/second)
result Code:
bit[1 .. 3] := false ;
foreach new state s do {
  // "first update the bits in a consistent order"
  bit'[1] := 'b(s) ;
  bit'[2] := 'f(s) \/ ! 'e(s) /\ bit[2] ;
  bit'[3] := 'd(s) /\ bit'[2] \/ 'c(s) /\ bit[3] ;
  // "then check whether the formula is violated"
  if 'a(s) /\ ! (bit[1] /\ bit'[3]) then
    Error ;
  // "finally, update the state of the monitor"
  bit[1 .. 3] := bit'[1 .. 3]
}

Maude>

```

The K Module

One should upload the next module whenever one wants to use the K technique to define a language, logic or tool. Note that the module below has nothing to do with our particular logic under consideration in this paper; that is the reason for which we exiled it here.

```

fmod K is
  sorts KConfigurationItem KConfiguration .
  subsort KConfigurationItem < KConfiguration .
  op empty : -> KConfiguration .
  op _ : KConfiguration KConfiguration -> KConfiguration [assoc comm id: empty] .

  sorts KComputationItem KNeComputation KComputation .
  subsort KComputationItem < KNeComputation < KComputation .
  op nil : -> KComputation .
  op _->_ : KComputation KComputation -> KComputation [assoc id: nil] .
  op _->_ : KNeComputation KNeComputation -> KNeComputation [ditto] .

  sort KComputationList .
  subsort KComputation < KComputationList .
  op nil : -> KComputationList .
  op _ , _ : KComputationList KComputationList -> KComputationList [assoc id: nil] .

  sort KResult KResultList .
  subsorts KResult < KResultList < KComputation .
  op nil : -> KResultList .

```

```

op __ : KResultList KResultList -> KResultList [assoc id: nil] .

op [_] : KComputationList -> KComputationItem .
op [_] : KResultList -> KComputationItem .
op [_|_] : KComputationList KResultList -> KComputationItem .

var K : KNeComputation . var Kl : KComputationList .
var R : KResult . var Rl : KResultList .
eq [K,Kl] = K -> [Kl | nil] .
eq R -> [K,Kl | Rl] = K -> [Kl | Rl,R] .
eq R -> [nil | Rl] = [Rl,R] .
endfm

```

To use K , after importing the module above, one should define one's own constructors for configuration items (sort `KConfigurationItem`), for computation items (sort `KComputationItem`), and for results (sort `KResult`). For our example, we defined all these at the beginning of the module `MONITOR-GENERATION`.

Exercises

Exercise 15 *Theorem 19 showed that only two temporal operators are actually needed. All the others can be regarded as derived operators. This is similar to how only two logical connectives are actually needed in propositional logic, say \neg and \wedge . Therefore, it is tempting to pick only two temporal operators and then desugar all the others to them. Explain the drawbacks of doing so in terms of the time/space complexity of the resulting monitors, addressing each of the discussed monitoring approaches separately: rewriting (Section 10.4), monitor generation (Section 10.5) including post-generation optimizations (Section 10.5.3), as well as directly-optimal monitor generation (Section 10.7).*

Exercise 16 *The monitor synthesis algorithms discussed in this chapter generate monitors whose states are vectors of bits, and which execute some simple code each time a new event is observed. Explain how you can generate conventional finite-state automata from such monitors. Explain the advantages and disadvantages of using automata generated this way as monitors.*

Chapter 11

Monitoring “Always-Past” Temporal Safety with Call-Return

Material from [229]

Abstract: We present an extension of past time LTL with call/return atoms, called PTCARET, together with a monitor synthesis algorithm for it. PTCARET includes abstract variants of past temporal operators, which can express properties over traces in which terminated function or procedure executions are abstracted away into a call and a corresponding return. This way, PTCARET can express safety properties about procedural programs which cannot be expressed using conventional linear temporal logics. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula has. The stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula has: push on begins, pop on ends of procedure executions. An optimized implementation is also discussed and is available to download.

11.1 Introduction

Theoretically speaking, it appears to be straightforward to monitor properties expressed as past time linear temporal logic (PTLTL) formulae, since the fix-point semantics of the temporal operators gives a direct deterministic automaton. The practical challenge in monitoring PTLTL formulae stays in how to do it *efficiently*, both time-wise and memory-wise, so that the added runtime overhead to the observed system is minimal. Since in a real-life runtime verification application there could be millions of monitor instances living at the same time, each observing tens of millions of events (see, e.g., [9, 25] and [58, 62] for numbers and evaluations of runtime verification systems on large benchmarks), every bit of memory or monitor processing time may translate into significantly higher runtime overhead, to an extent that the overall use of runtime verification in a particular application may become unfeasible. For example, in many cases it may not be a good idea to generate an actual deterministic automaton as a monitor, because that may have an exponential or worse size; instead, a non-deterministic automaton performing an NFA-to-DFA construction on the fly saving space exponentially may be more appropriate, or even a monitor that does not store any automaton at all, but has an efficient way to generate the next state on-the-fly.

Havelund and Roşu proposed a monitor synthesis algorithm for past-time LTL (PTLTL) formulae φ [129]. The generated monitors implement the recursive semantics of PTLTL using a dynamic programming technique, and need $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Roşu proposed an improved monitor synthesis algorithm for PTLTL in [220] (unpublished technical report) which, using a divide-and-conquer strategy, generates monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ .

Alur *et al.* gave an extension of linear temporal logic (LTL) with calls and returns [12], called CARET. Unlike LTL, CARET allows for matching call/return states in linear traces, allowing to express program trace properties not expressible using plain LTL. In particular, one can express properties on the execution stack of a program, such as “function g is always called from within function f ”, or structured-programming safety policies such as “each method must release before it terminates all the locks that it acquired during its execution”, or even properties that are allowed to be temporarily violated, such as “user u never directly accesses the passwords file (but may access it through system procedures)”. Because of allowing such

important and desirable safety properties to be formally stated at the same time faithfully including LTL, CARET can be a more attractive temporal logic than LTL, provided of course that the complexity of checking programs against CARET formulae does not make it unfeasible.

We define a past time variant of CARET, called PTCARET, show by examples its usefulness in expressing a series of safety properties involving calls of functions/procedures, and then propose a monitor synthesis algorithm for properties expressed as PTCARET formulae. Motivated by practical reasons, PTCARET distinguishes call/return states from begin/end states: the former take place in the caller’s context, while the latter take place in the callee’s. This simple and standard distinction allows more flexibility and elegance in expressing properties, but requires an additional (but reasonable) constraint on traces: calls always immediately precede begins, and ends always immediately precede returns.

PTCARET conservatively extends PTLTL by adding abstract variants of temporal operators, namely “abstract previously” and “abstract since”. The semantics of these operators is that of their corresponding core PTLTL operators “previously” and “since”, but on the *abstract* trace obtained by collapsing executed functions or procedures into only two states, namely the caller’s state at the call of the invoked function or procedure and the caller’s state at its corresponding return. In other words, from the point of view of the abstract temporal operators, the intermediate states generated during function executions are invisible. Of course, the standard temporal operators continue to “see” the whole trace.

The monitors generated from PTCARET formulae using the proposed algorithm have both a monitor state and a monitor stack, so they can be regarded as push-down automata; however, both the monitor states and the data pushed onto stacks are calculated online, on a by-need basis. The monitor state is encoded on as many bits as standard past time operators in the original formula, while the monitor stack pushes/pops as many bits of data as abstract temporal operators in the original formula. If no abstract temporal operators are used in a PTCARET formula, that is, if the PTCARET formula is a PTLTL formula, then its generated monitor is identical to that obtained using the technique in [220]. In other words, not only is PTCARET a conservative extension of PTLTL, but the proposed monitor synthesis algorithm conservatively extends the best known, provably optimal monitor synthesis algorithm for PTLTL.

The proposed PTCARET monitor synthesis algorithm has been imple-

mented and is available to download and experiment with via a web interface at [20]. The rest of the paper is structured as follows: Section 11.2 discusses PTCARET as an extension of PTLTL; Section 15.2 introduces useful derived operators and shows some examples of PTCARET specifications. Section 11.4.2 discusses our monitor synthesis algorithm, including its implementation. Section 11.5 concludes the paper.

11.2 PTLTL and PTCARET

We here recall past time linear temporal logic (PTLTL) and define its extension PTCARET. For simplicity, we assume only two types of past operators, namely “previously” and “since”. Other common or less common temporal operators can be added as derived operators. PTLTL contains only the usual, standard variants of temporal operators, while PTCARET contains both standard and abstract variants. We follow the usual recursive semantics of past time LTL and adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [129], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring and can easily be accommodated.

Definition 46 *Syntactically, PTLTL consists of formulae over the grammar*

$$\varphi ::= \text{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \odot\varphi \mid \varphi \mathcal{S} \varphi,$$

where a ranges over a set A of state predicates. Other common syntactic constructs can be defined as derived operators in a standard way: *false* is $\neg\text{true}$, $\diamond\varphi$ (“eventually in the past”) is $\text{true}\mathcal{S}\varphi$, $\Box\varphi$ (“always in the past”) is $\neg(\diamond\neg\varphi)$, etc.

LTL’s models, even for its safety fragment, traditionally are *infinite traces* (see, e.g., [187]), where a trace is a sequence of *states*, where a state is commonly abstracted as a set of atomic predicates in A .

refer to Chapter 3 below; or merge some of the text earlier in the book; or even remove it altogether.

According to Lamport [180], a *safety property* is a set of such infinite traces (properties are commonly identified with the sets of traces satisfying them)

such that once an execution “violates” it then it can never satisfy it again later. Formally, a set of infinite traces Q is a safety property if and only if for any infinite trace u , if $u \notin Q$ then there is some finite prefix w of u such that $wv \notin Q$ for all infinite traces v .

It can be shown that there are as many safety properties as real numbers [220]. Unfortunately, any logical formalism can define syntactically only as many formulae as natural numbers. Thus, any logical formalism can only express a small portion of safety properties. In LTL, a common way to specify safety properties is as “always past” formulae, that is, as formulae of the form $\Box\varphi$ (\Box is “always in the future”), where φ is a formula in PTLTL. There are two problems with identifying the problem of monitoring a PTLTL specification φ with checking the running system against the LTL safety formula $\Box\varphi$: on the one hand, LTL has an infinite trace semantics, while during monitoring we only have a finite number of past states available, and, on the other hand, once the LTL formula $\Box\varphi$ is violated then it can never be satisfied in the future. However, a major use of monitoring is in the context of recoverable systems, in the sense that the monitor can trigger recovery code when φ is violated, in the hope that φ will be satisfied from here on. For these reasons, we adopt a slightly modified semantics of past time LTL, namely the one on finite traces borrowed from [129]:

Definition 47 *A (program) state is a set of atomic predicates in A ; let s, s' , etc., denote states, and let $ProgState$ denote the set of all states. A trace is a finite sequence of states in $ProgState^*$; let w, w' , etc., denote traces, and ϵ denote the empty trace. If $w \neq \epsilon$, that is, if $w = w's$ for some trace w' and some state s , then we let $prefix(w)$ denote the trace w' and call it the (concrete) prefix of w , and let $last(w)$ denote the state s . The satisfaction relation $w \models \varphi$ between a trace w and a PTLTL formula φ is defined recursively as follows:*

$w \models true$		<i>is always true,</i>
$w \models a$	<i>iff</i>	$w \neq \epsilon$ and $a \in last(w)$,
$w \models \neg\psi$	<i>iff</i>	$w \not\models \psi$,
$w \models \psi \wedge \psi'$	<i>iff</i>	$w \models \psi$ and $w \models \psi'$,
$w \models \odot\psi$	<i>iff</i>	$w \neq \epsilon$ and $prefix(w) \models \psi$,
$w \models \psi \mathcal{S} \psi'$	<i>iff</i>	$w \neq \epsilon$ and ($w \models \psi'$ or $w \models \psi$ and $prefix(w) \models \psi \mathcal{S} \psi'$).

Unlike other places in the book, previously is defined with the “strong” meaning above.

We next introduce PTCARET as an extension of PTLTL. Syntactically, it only adds abstract versions of the two temporal operators “previously” and “since” to PTLTL; semantically, some special atomic predicates corresponding to calls, returns, begins and ends of functions/procedures need to be assumed, as well as some natural and practically reasonable restrictions on traces.

Definition 48 PTCARET *syntactically extends* PTLTL *with:*

$$\begin{array}{lcl} \varphi & ::= & \dots \\ & | & \overline{\varphi} \\ & | & \varphi \mathcal{S} \varphi \end{array}$$

The former is called “abstract previously” and the latter “abstract since”.

The semantics of abstract previously and since are defined exactly as the semantics of their concrete counterparts, but on an abstract version of the trace from which all the intermediate states of the terminated function or procedure executions are erased. In order for this erasure, or abstraction, process to work, we need to impose some constraints on traces that are always satisfied in practice.

Definition 49 In PTCARET, the set of atomic predicates A contains four special predicates: *call*, *begin*, *end*, and *return*. A state contains at most one of these and is called *call*, *begin*, *end*, or *return* state if it contains the corresponding predicate. PTCARET traces are constrained to the following restrictions:

- (1) any call state, except when the last one, must be immediately followed by a begin state, and any begin state must be immediately preceded by a call state;
- (2) any end state, except when the last one, must be immediately followed by a return state, and any return state must be immediately preceded by an end.

For a trace w as above, we let \overline{w} denote its abstraction, which is obtained by iteratively erasing contiguous subtraces $s_b w' s_e$ of w in which s_b is a begin state, s_e is an end state which is not the last one in w , and w' contains no begin or end states. One more restriction is imposed on PTCARET traces:

- (3) the abstractions of PTCARET traces contain no return states which are not immediately preceded by call states.

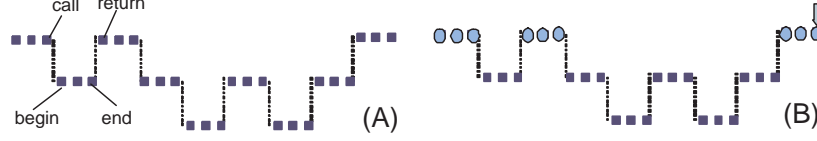


Figure 11.1: PTCARET trace (A) and abstraction (B). \Downarrow : end of w , \blacksquare : w state, \circ : \bar{w} state.

Call and return states occur in the caller’s context. Thus, call/return states can contain other predicates which may not be possible to evaluate in the callee’s context during runtime monitoring. The begin/end states are generated in the callee’s context, at the beginning and at the end of the execution of the invoked function, respectively. Similarly, for some common programming languages, begin/end states may contain other predicates that cannot be evaluated in the caller’s context. The original CARET logic [12] did not distinguish between call and begin states or between end and return states. We included all four of them in PTCARET for the reasons above and also because most trace monitoring systems (e.g., Tracematches [9, 25] and MOP [58, 62]) make a clear distinction between these four types of states.

Fig. 11.1 (A) shows a PTCARET trace. To better reflect the call-return structure of the PTCARET trace, states are placed on different levels: states on the higher level are generated in the caller’s context while those on the lower level are generated in the callee’s. The vertical dotted lines connect the corresponding call-begin and end-return pairs. Fig. 11.1 (B) shows the abstraction of that trace: if w ends with the state pointed by \Downarrow , \bar{w} contains only the circled states.

Restrictions (1) and (2) on PTCARET traces are very natural. One source of doubt though can be the sub-requirements that any return state must be preceded by an end state, and that any begin state must be preceded by a call state. While a return or a begin can indeed happen in any programming language only after a corresponding end or call state, respectively, one may argue that monitoring of a property should be allowed to start at any moment, in particular in between call and begin, or in between end and return states. While our synthesized monitors from PTCARET formulae (see Section 11.4.2) can be easily adapted to start monitoring at any moment in the trace, for the sake of a smoother and simpler development of the theoretical foundations of PTCARET, we assume that any PTCARET trace

starts from the beginning of the program execution and thus satisfies the above-mentioned restrictions. Restriction (3) ensures that a trace does not contain return states that do not have corresponding matching call states, also a natural restriction on complete traces.

Our definition of trace abstraction above is admittedly operational, but we think that it captures the desired end/begin matching concept both compactly and intuitively. Alternatively, we could have followed the CARET style in [12] and define the matching begin state of an end state as the latest begin state containing a balanced number of begin/end states in between.

uncomment the next text?

Definition 50 For a non-empty PTCARET trace w , let $\overline{prefix}(w)$, called the *abstract prefix* of w (not to be confused with the abstraction of the prefix of w , $\overline{prefix(w)}$), be either $prefix(w)$ if $last(w)$ is not a return state, or otherwise the prefix of w up to and including the corresponding matching call state of $last(w)$ if it is a return state; formally, if $last(w)$ is a return state then $\overline{prefix}(w)$ is the trace $w's_c$, where $w = w'w''$ for some w'' with $\overline{w''} = s_cs_r$, where s_c and s_r are call and return states, respectively.

Fig. 11.2 illustrates $\overline{prefix}(w)$ on two traces, with the down arrow pointing to the ends of the traces. In Fig. 11.2 (A) we assume that w ends with a state that is not a **return** (the arrow points to a **call** state) and in Fig. 11.2 (B) w ends with a **return** state (the states of the corresponding $\overline{prefix}(w)$ are marked with diamonds).

Definition 51 The satisfaction relation between a PTCARET trace w and a PTCARET formula φ is defined recursively exactly like in PTLTL for the PTLTL operators, and as follows for the two abstract temporal operators:

$$\begin{aligned} w \models \overline{\circ}\psi & \quad \text{iff} \quad w \neq \epsilon \text{ and } \overline{prefix}(w) \models \psi, \\ w \models \psi \overline{\mathcal{S}}\psi' & \quad \text{iff} \quad w \neq \epsilon \text{ and } (w \models \psi' \text{ or } w \models \psi \text{ and } \overline{prefix}(w) \models \psi \overline{\mathcal{S}}\psi'). \end{aligned}$$

Therefore, a formula $\overline{\circ}\psi$ is satisfied in a return state iff ψ was satisfied at the corresponding matching call state. It is satisfied in a non-return state, including an end state, iff $\overline{\circ}\psi$ is satisfied in that state (that is, if and only if ψ was satisfied in the concrete (non-abstract) previous state).

Fig. 11.3 compares the \circ and $\overline{\circ}$ operators. The arrows point, for each state, where the formula ψ in $\circ\psi$ (A) and in $\overline{\circ}\psi$ (B) holds. For most states,

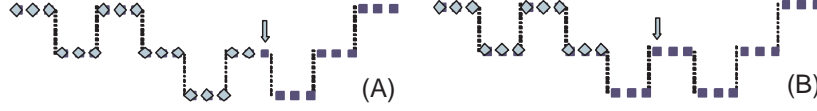


Figure 11.2: $\overline{prefix}(w)$ on two traces, (A) and (B). \Downarrow : the end of w , \diamond : state in $\overline{prefix}(w)$.

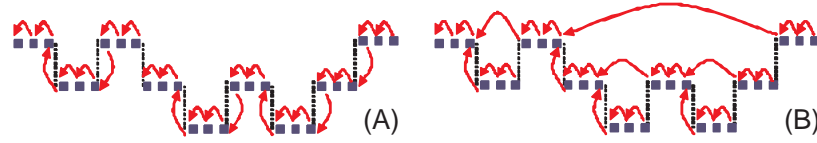


Figure 11.3: Concrete (A) and abstract (B) “previous” states for \circ and $\bar{\circ}$.

their abstract previous state is the concrete previous one; the only difference is on return states, because the abstract previous state of a return state is its call state.

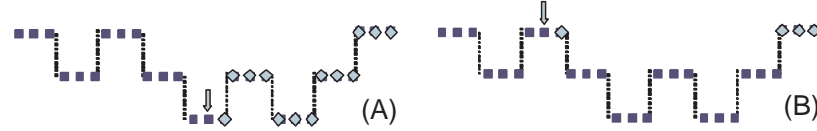


Figure 11.4: $\psi \mathcal{S} \psi'$ (A) versus $\psi \bar{\mathcal{S}} \psi'$ (B). \Downarrow : where ψ' holds, \diamond : where ψ holds.

Figure 11.4 compares $\psi \mathcal{S} \psi'$ and $\psi \bar{\mathcal{S}} \psi'$. Notice that the various call/return levels play no role in the satisfaction of $\psi \mathcal{S} \psi'$, but that they play a crucial role in the satisfaction of $\psi \bar{\mathcal{S}} \psi'$: for the latter, ψ' must hold on the same level or a higher level as the level of the current state. One can show the following expected property of abstract since:

Proposition 17 $\varphi_1 \bar{\mathcal{S}} \varphi_2$ is semantically equivalent to $\varphi_2 \vee \varphi_1 \wedge \bar{\circ}(\varphi_1 \bar{\mathcal{S}} \varphi_2)$.

One should not get tricked and assume that $w \models \bar{\circ}\varphi$ if and only if $\bar{w} \models \circ\varphi$, or that $w \models \varphi_1 \bar{\mathcal{S}} \varphi_2$ if and only if $\bar{w} \models \varphi_1 \mathcal{S} \varphi_2$! The reason is that subformulae φ , φ_1 or φ_2 may contain concrete temporal operators whose

semantics still involve the entire execution trace, not only the abstract one. Some examples in this category are shown in Section 15.2. Nevertheless, the following holds:

Proposition 18 *For a PTCARET trace w and formula φ containing no concrete temporal operators \circ and \mathcal{S} , $w \models \varphi$ iff $\bar{w} \models \hat{\varphi}$, where $\hat{\varphi}$ is the PTLTL formula replacing each abstract temporal operator in φ by its concrete variant.*

11.3 PTCARET Derived Operators and Examples

Besides the usual derived Boolean operators and past time temporal operators “eventually in the past”, “always in the past”, as well as “start”, “stop”, and “interval” operators like in [129], which can all be also defined abstract variants, we can define several other interesting, PTCARET-specific derived operators. In the rest of the paper we use the standard notation for the derived Boolean operators, e.g., “ \rightarrow ”, “ \vee ”, etc., with their usual precedences, and assume that “ \circ ” binds as tight as “ \neg ” while “ \mathcal{S} ” binds tighter than the binary Boolean operators.

At beginning. Suppose that one would like a particular property, say ψ , to hold at the beginning of the execution of the current function. We can define the derived temporal operator $@_b$, say “at beginning”, as follows:

$$@_b\psi \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \wedge (\neg\text{begin} \rightarrow \circ(\text{begin} \rightarrow \psi)\overline{\mathcal{S}}\text{begin}).$$

Note that the concrete “previously” operator is used inside the argument of the “abstract since” operator. The above is correct because the last begin state seen by the “abstract since” is indeed the beginning of the current function or procedure. One should not get tricked and try to define the above as:

$$@_b\psi \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \wedge (\neg\text{begin} \rightarrow (\text{begin} \rightarrow \psi)\overline{\mathcal{S}}\text{call}).$$

That is because the current function may have called and returned from several other functions, and the “abstract since” can still see all the call/return states. The above would vacuously hold in such a case.

At call. Suppose now that one wants ψ to hold at the state when the current function was called. For the same reason as above, one cannot simply replace **begin** by **call** in the definition of $@_b$ above. However, one can define



Figure 11.5: Derived operators. \Downarrow : current state, \diamond : states for $@_b, \mathcal{S}_b$; \circ : states for $@_c, \mathcal{S}_c$

the derived temporal operator $@_c$, say “at call”, in terms of “at beginning” simply as follows:

$$@_c\psi \stackrel{\text{def}}{=} @_b\circ\psi.$$

In Fig. 11.5 (A), supposing that the current state is the one pointed to by the arrow, ψ should hold in the diamond state for $@_b\psi$ and in the circle state for $@_c\psi$.

Stack since on beginnings. The “abstract since” can be used to write properties in which the terminated function executions are irrelevant. There may be cases in which one wants to write properties referring exclusively to the execution stack of a program, ignoring any other states. For example, one may want to say that ψ held on the stack since property ψ' held. As usual, one may be interested in properties ψ and ψ' to hold either at call time, or at execution beginning time. Let us first define a “stack since on beginnings” derived operator:

$$\psi \overline{\mathcal{S}_b} \psi' \stackrel{\text{def}}{=} (\text{begin} \rightarrow \psi) \overline{\mathcal{S}} (\text{begin} \wedge \psi').$$

Stack since on calls. To define a “stack since on calls” one cannot simply replace *begin* by *call* in the above. Instead, one can define it as follows:

$$\varphi_1 \overline{\mathcal{S}_c} \varphi_2 \stackrel{\text{def}}{=} (\text{call} \rightarrow \varphi_1) \overline{\mathcal{S}} (\text{begin} \wedge \circ\varphi_2).$$

In Fig. 11.5 (B), if the current state is the one pointed by the arrow, the *begin* stack consists of the diamonds and the *call* stack consists of the circles.

With the stack since derived temporal operators above, one can further define other derived operators, such as “stack eventually in the past on calls” (say $\overline{\diamond_c}$), “stack always in the past on beginnings” (say $\overline{\Box_b}$), etc.

Let us next further illustrate the strength of PTCARET by specifying some concrete properties that would be hard or impossible to specify in PTLTL.

Suppose that in a particular context, function f must be called only directly by function g . Assuming $call_f$ and $call_g$ are predicates that hold when f and g are called, respectively, we can specify this property in PTCARET as follows:

$$call_f \rightarrow @_c call_g.$$

Suppose now that f can be called only directly or indirectly by g : a call to g must be on the stack whenever f is called. We can specify that as follows:

$$call_f \rightarrow \overline{\diamond}_c call_g.$$

A common safety property in many systems is that resources acquired during a function execution must be released before the function ends. Assuming that *acquire* and *release* are predicates that hold when the resource of interest is acquired or released, respectively, we can specify this property as follows:

$$\text{end} \rightarrow (\neg \text{acquire} \overline{S} \text{begin} \vee \neg(\neg \text{release} \overline{S} \text{acquire})).$$

A more complex example is discussed in Section 16.7.

11.4 A Monitor Synthesis Algorithm for PTCARET

As discussed in [220] for PTLTL, thanks to the recursive nature of the satisfaction relation on the standard PTLTL temporal operators (see Definition 47), the monitor generated from a PTCARET formula needs only one global bit per standard (non-abstract) temporal operator. This bit maintains the satisfaction status of the subformula corresponding to that standard temporal operator; when a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 47 and the bit is updated. In order for this to work, one needs to have already updated or have an easy way to calculate the status of the subformulae.

The situation is more complex for the abstract temporal operators, as one needs to store enough information about the past so that one is able to update the status of abstract operators' satisfaction regardless of how the future evolves. The main complication comes from the fact that one needs to “freeze” the satisfaction status of the subformulae corresponding to abstract

temporal operators whenever a begin state is observed, and then “unfreeze” it when the corresponding end state is observed, thus recovering the information that was available right when the function call took place. Fortunately, that can be obtained by using a stack to push/pop the satisfaction status of the abstract temporal subformulae.

More precisely, a stack bit is needed per abstract temporal operator in the PTCARET formula, maintaining the satisfaction status of the subformula corresponding to that abstract operator. When a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 50 and the stack bit updated; if the newly observed state is a begin, then the status of the stack bits is pushed on the stack *before* the actual monitor state update; if the newly observed state is an end, then the status of the stack bits is popped from the stack *after* the monitor state update.

11.4.1 The Target Language

To state and prove the correctness of any program generation algorithm, one needs to have a formal semantics of the target language. This section gives a formal syntax and semantics to the simple and generic language in which we synthesize monitors. One can very easily translate this language into standard languages, such as C, C++, C#, Java, or even into native machine code. For each PTCARET formula φ , we are going to generate (in Section 11.4.2) a monitor \mathcal{M}_φ as a statement in a language \mathcal{L}_φ . The only difference between the languages \mathcal{L}_φ is the set of variables that one can assign values to; the rest of the language constructs are the same for all φ . The language \mathcal{L}_φ has the following simple syntax (note that $\mathcal{L}_{\varphi_1} \subseteq \mathcal{L}_{\varphi_2}$ whenever φ_1 is a subformula of φ_2):

$$\begin{array}{ll}
Var & ::= \alpha_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \odot \text{ or } \mathcal{S}) \\
& \quad | \beta_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \odot \text{ or } \overline{\mathcal{S}}) \\
Exp & ::= true \mid A \mid Var \mid \neg Exp \mid Exp \wedge Exp \\
Stm & ::= Var := Exp \mid \text{if begin then push} \mid \text{if end then pop} \\
& \quad | \text{output}(Exp) \mid Stm \ Stm
\end{array}$$

Therefore, programs in \mathcal{L}_φ can use predicates in A (the atomic predicate set of PTCARET) as ordinary (Boolean) expressions, together with Boolean

variables α_ϕ and β_ϕ , one per standard and abstract temporal operator in φ , respectively, and together with Boolean constructs such as complement and conjunction. Statements can be composed using juxtaposition, and can be: α_ϕ or β_ϕ variable assignment, output of a Boolean expression, or conditional push/pop, the latter pushing or popping, by convention, precisely the bit vector $\vec{\beta}$. We assume a (rather conventional) denotational semantics for \mathcal{L}_φ as follows:

Definition 52 *If φ has k_1 standard temporal operators and k_2 abstract temporal operators, then let $MonState_\varphi$ (we think of \mathcal{L}_φ programs as monitors) be the state space of \mathcal{L}_φ , that is, the domain $Bool^{k_1} \times Bool^{k_2} \times Stack \times Output$, where $Bool$ is the set $\{true, false\}$, $Stack$ is the domain $(Bool^{k_2})^*$ of stacks, or lists, over bit vectors of size k_2 , and $Output$ is the domain $Bool^*$ of bit lists. Let the functions*

$$\begin{aligned} \llbracket - \rrbracket &: Exp \rightarrow MonState_\varphi \rightarrow ProgState \rightarrow Bool \\ \llbracket - \rrbracket &: Stm \rightarrow MonState_\varphi \rightarrow ProgState \rightarrow MonState_\varphi \end{aligned}$$

be defined as follows:

$$\begin{aligned} \llbracket true \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= true, \quad \llbracket a \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) = s(a), \\ \llbracket \alpha_\phi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \vec{\alpha}(i), \text{ where } i \leq k_1 \text{ is the } \vec{\alpha}\text{-index corresponding to } \phi, \\ \llbracket \beta_\phi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \vec{\beta}(j), \text{ where } j \leq k_2 \text{ is the } \vec{\beta}\text{-index corresponding to } \phi, \\ \llbracket b_1 \wedge b_2 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \llbracket b_1 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) \text{ and } \llbracket b_2 \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s), \\ \llbracket \alpha_\phi := b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}[\vec{\alpha}(i) \leftarrow \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)], \vec{\beta}, \sigma, \omega), \\ \llbracket \beta_\phi := b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}, \vec{\beta}[\vec{\beta}(j) \leftarrow \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)], \sigma, \omega), \\ \llbracket if \text{ begin then push} \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \begin{cases} (\vec{\alpha}, \vec{\beta}, \vec{\beta} \cdot \sigma, \omega) & \text{if } s(\text{begin}), \\ (\vec{\alpha}, \vec{\beta}, \sigma, \omega) & \text{otherwise,} \end{cases} \\ \llbracket if \text{ end then pop} \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \begin{cases} (\vec{\alpha}, \vec{\beta}', \sigma', \omega) & \text{if } s(\text{end}) \text{ and } \sigma = \vec{\beta}' \cdot \sigma', \\ (\vec{\alpha}, \vec{\beta}, \sigma, \omega) & \text{otherwise,} \end{cases} \\ \llbracket output(b) \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= (\vec{\alpha}, \vec{\beta}, \sigma, \omega \cdot \llbracket b \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)), \\ \llbracket stm \text{ } stm' \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) &= \llbracket stm' \rrbracket(\llbracket stm \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s)). \end{aligned}$$

We can now associate a function $\llbracket \mathcal{M}_\varphi \rrbracket : MonState_\varphi \rightarrow ProgState \rightarrow MonState_\varphi$ to each program \mathcal{M}_φ in \mathcal{L}_φ . For a monitor state $(\vec{\alpha}, \vec{\beta}, \sigma, \omega) \in MonState_\varphi$ and a program state $s \in ProgState$, $\llbracket \mathcal{M}_\varphi \rrbracket(\vec{\alpha}, \vec{\beta}, \sigma, \omega)(s) = (\vec{\alpha}', \vec{\beta}', \sigma', \omega')$ if and only if the monitor \mathcal{M}_φ executed in state $(\vec{\alpha}, \vec{\beta}, \sigma, \omega)$ when program state s is observed, produces monitor state $(\vec{\alpha}', \vec{\beta}', \sigma', \omega')$.

Definition 53 *By abuse of notation, we also let $\llbracket \mathcal{M}_\varphi \rrbracket : \text{ProgState}^* \rightarrow \text{MonState}_\varphi$ be the function (false^{k_1} is the vector of k_1 false bits, and ϵ is the empty list):*

$$\begin{cases} \llbracket \mathcal{M}_\varphi \rrbracket(\epsilon) = (\text{false}^{k_1}, \text{false}^{k_2}, \epsilon, \epsilon) & \text{— the “initial” monitor state —} \\ \llbracket \mathcal{M}_\varphi \rrbracket(ws) = \llbracket \mathcal{M}_\varphi \rrbracket(\llbracket \mathcal{M}_\varphi \rrbracket(w))(s) \end{cases}$$

11.4.2 The Monitor Synthesis Algorithm

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as in [129], because different applications may choose different implementation paradigms. For example, our implementation of the PTCARET logic plugin in the context of the MOP system [58, 62], discussed in Section 16.7, uses term rewriting techniques. The monitoring code for a PTCARET formula φ can be split into three pieces: code to be executed before the monitor outputs the satisfaction status of the formula, the outputting code, and code to be executed after the output. Let $\text{Code}_{\text{before}}^\varphi$ denote the former and let $\text{Code}_{\text{after}}^\varphi$ denote the latter.

$\text{Code}_{\text{before}}^\varphi$ is concerned with updating the status of the “since” operators in a bottom-up fashion, while $\text{Code}_{\text{after}}^\varphi$ with updating the status of the “previously” operators. Indeed, in order to output the satisfaction status of φ , one needs to know the status of all the “since” operators, which may depend upon values in the current state as well as upon values of nested “since” operators, so the inner “since” operators need to be processed before the outer ones. On the other hand, one need not know the particular details (values of atomic predicates) of the current state in order to know the status of the “previously” operators; all one needs to make sure of is that the status of the “previously” operators has been updated at the appropriate previous state (or states in the case of “abstract previously”), after the monitor output. Interestingly, note that, unlike the “since” operators, the “previously” operators need to be processed in a top-down fashion, that is, the outer ones need to be processed before the inner ones.

Note that the monitors \mathcal{M}_φ generated in Figure 11.6 are well-defined, in the sense that each time a generated Boolean expression $\bar{\psi}$ is executed, all the α and β bits that are needed have been calculated. That is because the code is generated following a DFS traversal of the original PTCARET formula. \mathcal{M}_φ is run at each newly generated event, or program state, and outputs either *true* or *false*. Note that each \mathcal{M}_φ has the form “(if begin then

INPUT: A PTCARET formula φ

OUTPUT: Code that monitors φ

Step 1 Allocate a bit α_ϕ , initially *false*, for each subformula ϕ of φ rooted in a standard temporal operator. Intuition for this bit is:

- if $\phi = \odot\psi$ then α_ϕ says if ψ was satisfied at the previous state;
- if $\phi = \psi \mathcal{S} \psi'$ then α_ϕ says if ϕ was satisfied at previous state.

Step 2 Allocate a bit β_ϕ , initially *false*, for each subformula ϕ of φ rooted in an abstract temporal operator. Intuition for this bit is:

- if $\phi = \overline{\odot}\psi$ then β_ϕ says if ψ was satisfied at abstract previous state;
- if $\phi = \psi \overline{\mathcal{S}} \psi'$, β_ϕ says if ϕ was satisfied at abstract previous state.

Step 3 Initialize $Code_{before}^\varphi$ and $Code_{after}^\varphi$ as follows:

- $Code_{before}^\varphi$ to the code “if begin then push”, and
- $Code_{after}^\varphi$ to the code “if end then pop”.

Notation: For subformulae ϕ of φ , let $\overline{\phi}$ be the Boolean expression replacing in ϕ each temporal-operator-rooted subformula ψ which is not a subformula of another temporal-operator-rooted subformula of ϕ , by either α_ψ when ψ is rooted in a standard temporal operator, or by β_ψ when ψ is rooted in an abstract operator. E.g., $a \wedge \odot b \overline{\mathcal{S}} c \wedge \odot (d \mathcal{S} \overline{\odot} e)$ is $a \wedge \beta_{\odot b \overline{\mathcal{S}} c} \wedge \alpha_{\odot (d \mathcal{S} \overline{\odot} e)}$.

Step 4 Following a depth-first-search (DFS) traversal of φ , for each subformula ϕ of φ rooted in a temporal operator do:

- if $\phi = \odot\psi$ then $Code_{after}^\varphi \leftarrow (\alpha_\phi := \overline{\psi}) \text{ } Code_{after}^\varphi$
- if $\phi = \overline{\odot}\psi$ then $Code_{after}^\varphi \leftarrow (\beta_\phi := \overline{\psi}) \text{ } Code_{after}^\varphi$
- if $\phi = \psi \mathcal{S} \psi'$ then $Code_{before}^\varphi \leftarrow Code_{before}^\varphi (\alpha_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_\phi)$
- if $\phi = \psi \overline{\mathcal{S}} \psi'$ then $Code_{before}^\varphi \leftarrow Code_{before}^\varphi (\beta_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \beta_\phi)$

Step 5 Output monitor \mathcal{M}_φ as the code “ $Code_{before}^\varphi \text{ output}(\overline{\varphi}) \text{ } Code_{after}^\varphi$ ”

Figure 11.6: The monitor synthesis algorithm for PTCARET

push) C_1^φ output(O^φ) C_2^φ (if end then pop)”, for some potential statements C_1^φ and C_2^φ , and for some Boolean expression O^φ . To simplify notation, we introduce the following:

Definition 54 Let $\langle C_1^\varphi, O^\varphi, C_2^\varphi \rangle$ be a shorthand for:

if begin then push
 C_1^φ
 output(O^φ)
 C_2^φ
 if end then pop

We use \emptyset for C_1^φ or C_2^φ when they do not exist.

The following result structurally relates monitors generated for formulae φ to monitors generated for its subformulae. One can use this proposition as an equivalent, recursive way to synthesize monitors for PTCARET:

Proposition 19 If $\mathcal{M}_\psi = \langle C_1^\psi, O^\psi, C_2^\psi \rangle$ and $\mathcal{M}_{\psi'} = \langle C_1^{\psi'}, O^{\psi'}, C_2^{\psi'} \rangle$ then:

- $\mathcal{M}_{true} = \langle \emptyset, true, \emptyset \rangle$
- $\mathcal{M}_a = \langle \emptyset, a, \emptyset \rangle$
- $\mathcal{M}_{\neg\psi} = \langle C_1^\psi, \neg O^\psi, C_2^\psi \rangle$
- $\mathcal{M}_{\psi \wedge \psi'} = \langle C_1^\psi \ C_1^{\psi'}, O^\psi \wedge O^{\psi'}, C_2^{\psi'} \ C_2^\psi \rangle$
- $\mathcal{M}_{\odot\psi} = \langle C_1^\psi, \alpha_{\odot\psi}, (\alpha_{\odot\psi} := \overline{\psi}) \ C_2^\psi \rangle$
- $\mathcal{M}_{\psi \mathcal{S} \psi'} = \langle C_1^\psi \ C_1^{\psi'} \ (\alpha_{\psi \mathcal{S} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_{\psi \mathcal{S} \psi'}), \alpha_{\psi \mathcal{S} \psi'}, C_2^{\psi'} \ C_2^\psi \rangle$
- $\mathcal{M}_{\overline{\odot}\psi} = \langle C_1^\psi, \beta_{\overline{\odot}\psi}, (\beta_{\overline{\odot}\psi} := \overline{\psi}) \ C_2^\psi \rangle$
- $\mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} = \langle C_1^\psi \ C_1^{\psi'} \ (\beta_{\psi \overline{\mathcal{S}} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi \overline{\mathcal{S}} \psi'}), \beta_{\psi \overline{\mathcal{S}} \psi'}, C_2^{\psi'} \ C_2^\psi \rangle$

To prove the correctness of our monitor synthesis algorithm, we need to show that after observing any sequence of program states w , a synthesized monitor \mathcal{M}_φ outputs the same result as the satisfaction status of $w \models \varphi$. Therefore, we need to define “the output of the monitor \mathcal{M}_φ after observing w ”:

Definition 55 Let $\llbracket \mathcal{M}_\varphi \rrbracket : \text{ProgState}^+ \rightarrow \text{Bool}$ be defined for each (non-empty) $w \in \text{ProgState}^+$ as $\llbracket \mathcal{M}_\varphi \rrbracket(w) = b$ iff $\llbracket \mathcal{M}_\varphi \rrbracket(w) = (\vec{\alpha}, \vec{\beta}, \sigma, \omega \cdot b)$. For uniformity, let us extend $\llbracket \mathcal{M}_\varphi \rrbracket$ to a function $\text{ProgState}^* \rightarrow \text{Bool}$ (as in Definitions 47 and 50):

- $\llbracket \mathcal{M}_\varphi \rrbracket(\epsilon) = \text{false}$ when $\varphi = a, \odot\psi, \psi \mathcal{S} \psi', \overline{\odot}\psi, \psi \overline{\mathcal{S}} \psi'$;
- $\llbracket \mathcal{M}_{\neg\psi} \rrbracket(\epsilon) = \neg \llbracket \mathcal{M}_\psi \rrbracket(\epsilon)$;
- $\llbracket \mathcal{M}_{\psi \wedge \psi'} \rrbracket(\epsilon) = \llbracket \mathcal{M}_\psi \rrbracket(\epsilon) \wedge \llbracket \mathcal{M}_{\psi'} \rrbracket(\epsilon)$.

Proposition 20 The following hold for any $w \in \text{ProgState}^*$:

- $\llbracket \mathcal{M}_{\text{true}} \rrbracket(w)$ is always true,
- $\llbracket \mathcal{M}_a \rrbracket(w)$ iff $w \neq \epsilon$ and $a \in \text{last}(w)$,
- $\llbracket \mathcal{M}_{\neg\psi} \rrbracket(w)$ iff not $\llbracket \mathcal{M}_\psi \rrbracket(w)$,
- $\llbracket \mathcal{M}_{\psi \wedge \psi'} \rrbracket(w)$ iff $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$,
- $\llbracket \mathcal{M}_{\odot\psi} \rrbracket(w)$ iff $w \neq \epsilon$ and $\llbracket \mathcal{M}_\psi \rrbracket(\text{prefix}(w))$,
- $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(w)$ iff $w \neq \epsilon$ and ($\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$ or $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(\text{prefix}(w))$),
- $\llbracket \mathcal{M}_{\overline{\odot}\psi} \rrbracket(w)$ iff $w \neq \epsilon$ and $\llbracket \mathcal{M}_\psi \rrbracket(\overline{\text{prefix}}(w))$,
- $\llbracket \mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} \rrbracket(w)$ iff $w \neq \epsilon$ and ($\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$ or $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi \overline{\mathcal{S}} \psi'} \rrbracket(\overline{\text{prefix}}(w))$).

Proof: The non-trivial ones are those for temporal operators. We only discuss $\overline{\mathcal{S}}$, because the others follow the same idea and are simpler. The monitors for $\psi \overline{\mathcal{S}} \psi'$, ψ , and ψ' , respectively, following the notations in Proposition 19 are:

	$\mathcal{M}_{\psi \overline{\mathcal{S}} \psi'}$	\mathcal{M}_ψ	$\mathcal{M}_{\psi'}$
1.	if begin then push	if begin then push	if begin then push
2.	$C_1^\psi \ C_1^{\psi'}$	C_1^ψ	$C_2^{\psi'}$
3.	$\beta_{\psi \mathcal{S} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi \mathcal{S} \psi'}$		
4.	output($\beta_{\psi \mathcal{S} \psi'}$)	output($\overline{\psi}$)	output($\overline{\psi'}$)
5.	$C_2^{\psi'} \ C_2^\psi$	C_2^ψ	$C_2^{\psi'}$
6.	if end then pop	if end then pop	if end then pop

Note that the property holds vacuously if $w = \epsilon$. Assume now that $w = w's$, for some $s \in \text{ProgState}$. An interesting and useful property of the generated

monitors is that their semantics is very modular, and that pushing or popping $\vec{\beta}$ does not affect the modular semantics. For example, note that C_1^ψ in $\mathcal{M}_{\psi \mathcal{S} \psi'}$ uses no variables defined in $C_1^{\psi'}$ or in $C_2^{\psi'}$, and the bit $\beta_{\psi \mathcal{S} \psi'}$ is only defined in line 3. and used in lines 3. and 4. This modularity guarantees that, if we were to output $\bar{\psi}$ or $\bar{\psi}'$ at line 3. or 4. in $\mathcal{M}_{\psi \mathcal{S} \psi'}$, then its output after processing w would be nothing but $\llbracket \mathcal{M}_\psi \rrbracket(w)$ or $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$, respectively. That means that the $\bar{\psi}$ and $\bar{\psi}'$ in the expression assigned to $\beta_{\psi \mathcal{S} \psi'}$ at line 4. when processing the last state in w are $\llbracket \mathcal{M}_\psi \rrbracket(w)$ and $\llbracket \mathcal{M}_{\psi'} \rrbracket(w)$, respectively. We claim that $\beta_{\psi \mathcal{S} \psi'}$ in the assigned expression at line 4. is $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(\overline{prefix(w)})$. There are two cases to analyze. (1) if s is not a return state, then $\beta_{\psi \mathcal{S} \psi'}$ was assigned at line 3. in the previous execution of the monitor, when processing the last state in w' , so it is nothing but $\llbracket \mathcal{M}_{\psi \mathcal{S} \psi'} \rrbracket(prefix(w))$; and (2) if s is a return state, then it means that the last state in w' was an end state, so the vector $\vec{\beta}$ was popped from the stack at the end of the previous step. The only thing left to note is that our **push** on begins and **pop** or ends correctly match **begin** and **end** states; this follows from the fact that we assume traces complete and well-formed (Definition 49). \square

Theorem 20 *The monitor synthesis algorithm in Figure 11.6 is correct, that is, for any PTCARET formula φ and for any $w \in ProgState^*$, $\llbracket \mathcal{M}_\varphi \rrbracket(w)$ iff $w \models \varphi$.*

Proof: Straightforward, by induction on both the structure of φ and the length of w , noticing that there is a one-to-one correspondence between the definition of satisfaction in Definitions 47 and 50, and the properties in Proposition 20. \square

11.4.3 Implementation as Logic Plugin, Optimizations, Example

MOP [58, 62] is a configurable runtime verification framework, in which specification requirements formalisms can be added modularly, by means of *logic plugins*. A logic plugin essentially encapsulates a monitor synthesis algorithm for a formalism that one can then use to specify properties of programs. The current JavaMOP tool has logic plugins for future time LTL, past time LTL, Allen algebra, extended regular expressions, JML, JASS. JavaMOP takes a Java application to be monitored and specifications using any of the included formalisms together with validation and/or violation

handlers (saying what to do if property validated or violated, in particular nothing), and then waves them together in a runtime verified application by first generating monitors for all the properties using their corresponding logic plugins, and then generating and compiling an AspectJ extension of the original program (runtime monitors are “aspects”). To maintain a reduced runtime overhead (shown on large benchmarks to be, on average, below 10%), MOP piggybacks monitor states onto object states.

The PTCARET MOP logic plugin. We implemented the PTCARET monitor synthesis algorithm in Section 11.4.2 as an MOP logic plugin. Our implementation can be found and experimented with online at [20]. Large-scale experiments are still to be performed; we are currently engineering the MOP system to allow monitor states to piggyback not only object states, but also the program stack. In short, our implementation uses *term rewriting* and the Maude system [73], and follows the monitor synthesis algorithm in Figure 11.6 and its “equivalent”, recursive formulation in Proposition 19. Implementations in other languages are obviously also possible; however, term rewriting proved to be an elegant means to synthesize monitors from logical formulae in several other contexts (the other MOP plugins, as well as in JPaX [223]), and so seems to be here.

Our implementation starts by defining the Boolean expressions as an algebraic specification using Maude’s mixfix notation (equivalent to context-free grammars); derived Boolean operators are also defined, together with several simplification rules ($\neg \text{true} = \text{false}$, etc.). Boolean expressions are imported both in the target language module and in the PTCARET module. Both the target language and the PTCARET modules are defined as algebraic signatures, enriched with structural equalities which turn into simplification rules when executed; this way, for example, each PTCARET derived operator is defined with one equation capturing its definition. Several other derived operators are defined in addition to those discussed in Section 11.3. The monitor generation module imports both the target language and the PTCARET modules, and adds two equations per temporal logic operator; e.g., the equations below process the “abstract since”:

```

eq form(F1 Sa F2) = [form(F1), form(F2)] -> Sa .
eq k([exp(B1),exp(B2)] -> Sa -> K) code(I,C1,C2) nextBeta(N)
  = k(exp(beta[N]) -> K) code(I beta[N] := false,
                                C1 beta[N] := B2 or B1 and beta[N], C2) nextBeta(N + 1) .

```

First equation says that subformulae should be processed first (DFS traversal). The second equation combines the codes generated from the subformulae as

shown in Proposition 19, appending the assignment for the corresponding bit to **C1**. Note that **C1** here accumulates the “code before” of both subformulae; in terms of Proposition 19, it is “ $C_1^{\psi} C_1^{\psi'}$ ”. **I** accumulates the monitor initialization code. Finally the optimizations below are implemented also as rewrite rules.

Optimizations. Term-rewriting-based code-generation algorithms can be easily extended with optimizations, because these can be captured as rewrite rules. We discuss some of the optimizations enabled in our implementation. First, we perform Boolean simplifications when calculating $\bar{\psi}$ to reduce runtime overhead ($\neg\neg\psi = \psi$, $\text{true} \wedge \psi = \text{true}$, etc.). Another immediate optimization is the following. The generated code originally has the form (see Fig. 11.6) “(if begin then push) C (if end then pop)”, for some code C . However, since a program state can only contain at most one of the special predicates, this can be optimized into (syntax of target language needs to be slightly extended):

```

if begin then {
    push
     $C[\text{begin} \leftarrow \text{true}, \text{end} \leftarrow \text{false}, \text{call} \leftarrow \text{false}, \text{return} \leftarrow \text{false}]$ 
    exit
}
if end then {
     $C[\text{begin} \leftarrow \text{false}, \text{end} \leftarrow \text{true}, \text{call} \leftarrow \text{false}, \text{return} \leftarrow \text{false}]$ 
    pop
    exit
}
 $C[\text{begin} \leftarrow \text{false}, \text{end} \leftarrow \text{false}]$ 

```

After the substitutions above, further Boolean simplifications may be triggered. Also, some assignments may become redundant, such as, for example, “**beta**[3] := **beta**[3]”; rules to eliminate such assignments are also given. A further optimization on the generated code is possible, but we have not implemented it yet: some subformulae can repeat in different parts of the original formula; the current implementation generates monitoring code for each repeating instance, which is redundant and can be reduced using a smarter optimization algorithm.

Example. We here show the monitor generated by our implementation for a more complex PTCARET specification. Suppose that a program carries

out a critical multi-phase task and the following safety properties must hold when execution enters the second phase:

1. Execution entered the first phase within the same procedure;
2. Resource acquired within same procedure since first phase must be released;
3. Caller of current procedure must have had approval for the second phase;
4. Task is executed directly or indirectly by the procedure *safe_exec*.

These can be captured as the following PTCARET formula:

$$\begin{aligned}
enter_phase_2 \rightarrow & \quad (\neg(\neg enter_phase_1 \overline{S} \text{begin}) \\
& \quad \wedge (\neg acquire \overline{S} enter_phase_1 \vee \neg(\neg release \overline{S} acquire)) \\
& \quad \wedge @_c(has_phase_2_pass) \\
& \quad \wedge \overline{\otimes}_b(safe_exec)
\end{aligned}$$

Our implementation generates the following monitor for this specification:

```

if begin then {
  push(beta);
  beta[0] := safe_exec or beta[0];
  beta[1] := enter_ph1 or not acquire and beta[1];
  beta[2] := acquire or not release and beta[2];
  beta[3] := true; beta[4] := true;
  output(not enter_ph2 or not beta[4] and alpha[0] and beta[0] and (not beta[2] or beta[1]));
  alpha[3] := true;
  alpha[2] := alpha[1];
  alpha[1] := has_ph2_pass;
  alpha[0] := has_ph2_pass;
  exit
}
if end then {
  beta[1] := enter_ph1 or not acquire and beta[1];
  beta[2] := acquire or not release and beta[2];
  beta[3] := beta[3] and (not alpha[3] or alpha[2]);
  beta[4] := not enter_ph1 and beta[4];
  output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
  alpha[3] := false;
  alpha[2] := alpha[1];
  alpha[1] := has_ph2_pass;
  alpha[0] := has_ph2_pass;
  pop(beta);
  exit
}
beta[1] := enter_ph1 or not acquire and beta[1];

```

```

beta[2] := acquire or not release and beta[2];
beta[3] := beta[3] and (not alpha[3] or alpha[2]);
beta[4] := not enter_ph1 and beta[4];
output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
alpha[3] := false;
alpha[2] := alpha[1];
alpha[1] := has_ph2_pass;
alpha[0] := has_ph2_pass

```

The formula contains derived operators, e.g., $@_c$, which are first expanded. The monitoring code uses four α bits and five β bits (the expanded formula contains four concrete temporal operators and five abstract ones). For example, $\boxtimes_b(\text{safe_exec})$ is expanded into $(\text{begin} \rightarrow \text{true})\overline{\mathcal{S}}(\text{begin} \wedge \text{safe_exec})$, which is then simplified to $\text{true}\overline{\mathcal{S}}(\text{begin} \wedge \text{safe_exec})$, equivalent to $\boxtimes(\text{begin} \wedge \text{safe_exec})$. **beta**[0] in the generated code is used to check this operation; it only needs to be updated at the **begin** state, where it becomes true if *safe_exec* holds.

11.5 Conclusion and Future Work

We presented the logic PTCARET and a monitor synthesis algorithm for it. PTCARET includes abstract variants of past temporal operators. It can express safety properties about procedural programs which cannot be expressed using conventional PTLTL. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula had, while the stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula had. An optimized implementation of the monitor synthesis algorithm has been organized as an MOP logic plugin, and is available to download from [20]. There is room for further optimizations of the generated code. An extensive evaluation of the effectiveness of PTCARET runtime verification on large programs needs to be conducted. On the theoretical side, it would be interesting to explore the relationship between our monitors generated for PTCARET and the nested word automata in [51]; [51] gives an operational monitoring language for nested words based on BLAST's specification language. In contrast, our language is declarative and an operational encoding synthesized automatically.

uncommend the next text

11.6 Auxiliary Material - not included in original paper

In this section we show via an example how to generate dynamic programming code for a concrete *ptLTL*-formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next subsection. Let $\uparrow p \rightarrow [q, \downarrow (r \vee s)]_s$ be the *ptLTL*-formula that we want to generate code for. The formula states: “whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s ”. The code translation depends on an enumeration of the subformulae of the formula that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, \dots, \varphi_8$ be such an enumeration:

$$\begin{aligned}\varphi_0 &= \uparrow p \rightarrow [q, \downarrow (r \vee s)]_s, \\ \varphi_1 &= \uparrow p, \\ \varphi_2 &= p, \\ \varphi_3 &= [q, \downarrow (r \vee s)]_s, \\ \varphi_4 &= q, \\ \varphi_5 &= \downarrow (r \vee s), \\ \varphi_6 &= r \vee s, \\ \varphi_7 &= r, \\ \varphi_8 &= s.\end{aligned}$$

Note that the formulae have here been enumerated in a post-order fashion. One could have chosen a breadth-first order, or any other enumeration, as long as the enumeration invariant is true.

The input to the generated program will be a finite trace $t = s_1 s_2 \dots s_n$ of n events. The generated program will maintain a state via a function $update : \mathbf{State} \times Event \rightarrow \mathbf{State}$, which updates the state with a given event.

In order to illustrate the dynamic programming aspect of the solution, one can imagine recursively defining a matrix $s[1..n, 0..8]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$. Then one can fill the table according to the recursive semantics of past time LTL as described in Subsection 10.3.3. This would be the standard way of regarding the above satisfaction problem as a dynamic programming problem. An important observation is, however, that, like in many other dynamic programming algorithms, one doesn't have to store the entire table $s[1..n, 0..8]$, which would be quite large in practice; in this case, one needs only $s[i, 0..8]$ and

$s[i-1, 0..8]$, which we'll write $now[0..8]$ and $pre[0..8]$ from now on, respectively. It is now only a relatively simple exercise to write up the following algorithm for checking the above formula on a finite trace:

```

State  $state \leftarrow \{\}$ ;

bit  $pre[0..8]$ ;

bit  $now[0..8]$ ;

INPUT: trace  $t = s_1s_2\dots s_n$ ;

/* Initialization of  $state$  and  $pre$  */

 $state \leftarrow update(state, s_1)$ ;

 $pre[8] \leftarrow s(state)$ ;

 $pre[7] \leftarrow r(state)$ ;

 $pre[6] \leftarrow pre[7] \text{ or } pre[8]$ ;

 $pre[5] \leftarrow \text{false}$ ;

 $pre[4] \leftarrow q(state)$ ;

 $pre[3] \leftarrow pre[4] \text{ and not } pre[5]$ ;

 $pre[2] \leftarrow p(state)$ ;

 $pre[1] \leftarrow \text{false}$ ;

 $pre[0] \leftarrow \text{not } pre[1] \text{ or } pre[3]$ ;

/* Event interpretation loop */

for  $i = 2$  to  $n$  do {

     $state \leftarrow update(state, s_i)$ ;

     $now[8] \leftarrow s(state)$ ;

     $now[7] \leftarrow r(state)$ ;

     $now[6] \leftarrow now[7] \text{ or } now[8]$ ;

```

```

    now[5] ← not now[6] and pre[6];
    now[4] ← q(state);
    now[3] ← (pre[3] or now[4]) and not now[5];
    now[2] ← p(state);
    now[1] ← now[2] and not pre[2];
    now[0] ← not now[1] or now[3];
    if now[0] = 0 then
        output(■property violated■);
        pre ← now;
    };

```

In the following we explain the generated program.

Declarations Initially a state is declared. This will be updated as the input event list is processed. Next, the two arrays *pre* and *now* are declared. The *pre* array will contain values of all subformulae in the previous state, while *now* will contain the value of all subformulae in the current state.

Initialization The initialization phase consists of initializing the *state* variable and the *pre* array. The first event s_1 of the event list is used to initialize the *state* variable. The *pre* array is initialized by evaluating all subformulae bottom up, starting with highest formula numbers, and assigning these values to the corresponding elements of the *pre* array; hence, for any $i \in \{0 \dots 8\}$ $pre[i]$ is assigned the initial value of formula φ_i . The *pre* array is initialized in such a way as to maintain the view that the initial state is supposed stationary before monitoring is started. This in particular means that $\uparrow p$ is false, as well as is $\downarrow (r \vee s)$, since there is no change in state (indexes 1 and 5). The interval operator has the obvious initial interpretation: the first argument must be true and the second false for the formula to be true (index 3). Propositions are true if they hold in the initial state (indices 2, 4, 7 and 8), and boolean operators are interpreted the standard way (indices 0, 6).

Event Loop The main evaluation loop goes through the event trace, starting from the second event. For each such event, the state is updated, followed by assignments to the *now* array in a bottom-up fashion similar to the initialization of the *pre* array: the array elements are assigned values from higher index values to lower index values, corresponding to the values of the corresponding subformulae. Propositional boolean operators are interpreted the standard way (indices 0 and 6). The formula $\uparrow p$ is true if p is true now and not true in the previous state (index 1). Similarly with the formula $\downarrow (r \vee s)$ (index 5). The formula $[q, \downarrow (r \vee s)]_s$ is true if either the formula was true in the previous state, or q is true in the current state, and in addition $\downarrow (r \vee s)$ is not true in the current state (index 3). At the end of the loop an error message is issued if *now*[0], the value of the whole formula, has the value 0 in the current state. Finally, the entire *now* array is copied into *pre*.

Given a fixed *ptLTL* formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the *ptLTL* formula. The memory required is constant, since the length of the two arrays is the size of the *ptLTL* formula. However, one may want to also include the size of the formula, say m , into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while memory required is $2 \cdot (m + 1)$ bits. The authors conjecture that it's hard to find an algorithm running faster than the above in practical situations, though some slight optimizations are possible (see Section 10.5.3).

11.6.1 The Algorithm Formalized

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from a *ptLTL*-formula. It takes as input a formula and generates a program as the one above, containing a “for” loop which traverses the trace of events, while validating or invalidating the formula. The generated program is printed using the function **output**, which takes one or more string or integer parameters which are concatenated in the output. This algorithm is designed to generate pseudocode, but it can easily be adapted to generate code in any imperative programming language:

INPUT: past time LTL formula φ

let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the subformulae of φ ;

```

output("State  $state \leftarrow \{\}$ ");
output("bit  $pre[0..m]$ ");
output("bit  $now[0..m]$ ");
output("INPUT: trace  $t = s_1 s_2 \dots s_n$ ");
output("/* Initialization of  $state$  and  $pre$  */");
output("state  $\leftarrow update(state, s_1)$ ");
for  $j = m$  downto 0 do {
    output("     $pre[$ ,  $j$ ,  $]$   $\leftarrow$  ");
    if  $\varphi_j$  is a variable then
        output( $\varphi_j$ , " $(state)$ ");
    if  $\varphi_j$  is true then output("true");
    if  $\varphi_j$  is false then output("false");
    if  $\varphi_j = \neg \varphi_{j'}$  then
        output("not  $pre[$ ,  $j'$ ,  $]$ ");
    if  $\varphi_j = \varphi_{j_1} \text{ op } \varphi_{j_2}$  then
        output("pre[ $, j_1, ]$  op pre[ $, j_2, ]$ ");
    if  $\varphi_j = \odot \varphi_{j_1}$  then
        output("pre[ $, j_1, ]$ ");
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output("pre[ $, j_1, ]$ ");
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output("pre[ $, j_1, ]$ ");
    if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
        output("pre[ $, j_2, ]$ ");

```

```

if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
    output("pre[" ,  $j_1$ , "]" or pre[" ,  $j_2$ , "]" );
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
    output("pre[" ,  $j_1$ , "]" and not pre[" ,  $j_2$ , "]" );
if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
    output("not pre[" ,  $j_2$ , "]" );
if  $\varphi_j = \uparrow \varphi_{j'}$  then output("false;");
if  $\varphi_j = \downarrow \varphi_{j'}$  then output("false;");
};
output("/ * Event interpretation loop * /");
output("for  $i = 2$  to  $n$  do {");
for  $j = m$  downto 0 do {
    output("    now[" ,  $j$ , "]"  $\leftarrow$  ");
    if  $\varphi_j$  is a variable then output( $\varphi_j$ , "(state);");
    if  $\varphi_j$  is true then output("true;");
    if  $\varphi_j$  is false then output("false;");
    if  $\varphi_j = \neg \varphi_{j'}$  then output("not now[" ,  $j'$ , "]" );
    if  $\varphi_j = \varphi_{j_1} \text{ op } \varphi_{j_2}$  then
        output("now[" ,  $j_1$ , "]" op now[" ,  $j_2$ , "]" );
    if  $\varphi_j = \odot \varphi_{j_1}$  then output("pre[" ,  $j_1$ , "]" );
    if  $\varphi_j = \diamond \varphi_{j_1}$  then
        output("pre[" ,  $j$ , "]" or now[" ,  $j_1$ , "]" );
    if  $\varphi_j = \Box \varphi_{j_1}$  then
        output("pre[" ,  $j$ , "]" and now[" ,  $j_1$ , "]" );

```

```

if  $\varphi_j = \varphi_{j_1} S_s \varphi_{j_2}$  then
  output("(pre",  $j$ , "]" and now",  $j_1$ , "]") or
    now",  $j_2$ , "];");

if  $\varphi_j = \varphi_{j_1} S_w \varphi_{j_2}$  then
  output("(pre",  $j$ , "]" and now",  $j_1$ , "]") or
    now",  $j_2$ , "];");

if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_s$  then
  output("(pre",  $j$ , "]" or now",  $j_1$ , "]") and
    not now",  $j_2$ , "];");

if  $\varphi_j = [\varphi_{j_1}, \varphi_{j_2}]_w$  then
  output("(pre",  $j$ , "]" or now",  $j_1$ , "]") and
    not now",  $j_2$ , "];");

if  $\varphi_j = \uparrow \varphi_{j'}$  then
  output("(now",  $j'$ , "]" and
    not pre",  $j'$ , "];");

if  $\varphi_j = \downarrow \varphi_{j'}$  then
  output("(not now",  $j'$ , "]" and
    pre",  $j'$ , "];");

};

output("    if now[0] = 0 then

      output(■property violated■);

    output("    pre  $\leftarrow$  now;");

output("}");

```

op is any binary propositional connective. Since we have already given a detailed explanation of the example in the previous section, we shall only give a very brief description of this algorithm.

The formula should be first visited top down to assign increasing numbers to subformulae as they are visited. Let $\varphi_0, \varphi_1, \dots, \varphi_m$ be the list of all

subformulae. Because of the recursive nature of *ptLTL*, this step ensures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i-1} \models \varphi_{j'}$ for all $j \leq j' \leq m$.

Before we generate the main loop, we should first generate code for initializing the array *pre*[0..*m*], basically giving it the truth values of the subformulae on the initial state, conceptually being an infinite trace with repeated occurrences of the initial state. After that, the generated main event loop will process the events. The loop body will update/calculate the array *now* and in the end will move it into the array *pre* to serve as basis for the next iteration. After each iteration *i*, *now*[0] tells whether the formula is validated by the trace $s_1s_2\dots s_i$.

Since the formula enumeration procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an *ptLTL* formula in linear time with the size of the formula. The boolean operations used above are usually very efficiently implemented on any microprocessor and the arrays of bits *pre* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated “for” loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine’s resources. Consequently, the generated code is expected to run very fast. We shall next illustrate how such optimizations can be part of the translation algorithm.

Exercises

Exercise 17 Suppose that the set *A* of state predicates in Definition 49 contains only the four special predicates *call*, *begin*, *end* and *return*. Give an example of a safety property over PTCARET traces that is not expressible using the PTCARETlogic formalism.

Exercise 18 Specify using PTCARET the following property: function *f* can only be called, directly or indirectly, via function *g*, and at most once. In other words, no invocation of *g* can result in invoking *f* twice, directly or indirectly, and an invocation of *f* can only happen from within an invocation of *g*, directly or indirectly.

Exercise 19 Consider the four derived operators in Section 11.3. Modify the PTCARET monitor synthesis algorithm in Section 11.4.2, as well as the related results that yield its correctness, to generate monitoring code directly

for these derived operators, without first desugaring them to the standard operators.

Exercise 20 *Following a similar idea as in Exercise 16, explain how to generate a push-down automaton from a PTCARET monitor generated using the technique discussed in this chapter. Explain also the advantages and disadvantages of using such a push-down automata as monitors.*

Chapter 12

Efficient Monitoring of Parametric Context-Free Patterns

Material from [196], which extends [57]

Abstract: Recent developments in runtime verification and monitoring show that parametric regular and temporal logic specifications can be efficiently monitored against large programs. However, these logics reduce to ordinary finite automata, limiting their expressivity. For example, neither can specify structured properties that refer to the call stack of the program. While context-free grammars (CFGs) are expressive and well-understood, existing techniques for monitoring CFGs generate large runtime overhead in real-life applications. This paper demonstrates that monitoring parametric CFGs is *practical* (with overhead on the order of 12% or lower in most cases). We present a monitor synthesis algorithm for CFGs based on an LR(1) parsing algorithm, modified to account for good prefix matching. In addition, a logic-independent mechanism is introduced to support matching against the suffixes of execution traces.

12.1 Introduction

Runtime verification (RV) is a relatively new formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against

its requirements at runtime, as in testing. A large number of runtime verification approaches and systems, including TemporalRover [88], JPaX [126], JavaMaC [166], Hawk/Eagle [83], Tracematches [9, 25], J-Lo [38], PQL [192], PTQL [118], MOP [62, 58], Pal [52], RuleR [31], etc., have been developed recently. In a runtime verification system, monitoring code is generated from the specified properties and integrated with the system one wishes to monitor. Therefore, a runtime verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a program instrumentor. The chosen specification formalism determines the expressivity of the runtime verification approach and/or system.

Monitoring safety properties is arbitrarily complex [235]. Recent developments in runtime verification, however, show that regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. As shown by a series of experiments in the context of Tracematches [25] and JavaMOP [62], parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 12% or lower. However, both regular expressions and temporal logics reduce to ordinary finite automata when monitored, so they have inherently limited expressivity. More specifically, most runtime verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such runtime verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. Examples of such structured safety properties include “a resource should be released in the same method which acquired it” or “a resource cannot be accessed if the unsafe method `foo` is in the current call stack”.

12.1.1 Example

An important and desirable category of properties that cannot be expressed using regular patterns is one in which pairs of events need to match each other, potentially in a nested way. For example, suppose that one prefers to use one’s own locking mechanism for thread synchronization. As usual, for multiple reasons including the allowance of re-entrant synchronized methods (in particular to support recursion), locks are allowed to be acquired and released multiple times by any given thread. However, the lock is effectively released, so that other threads can acquire it, only when the lock releases

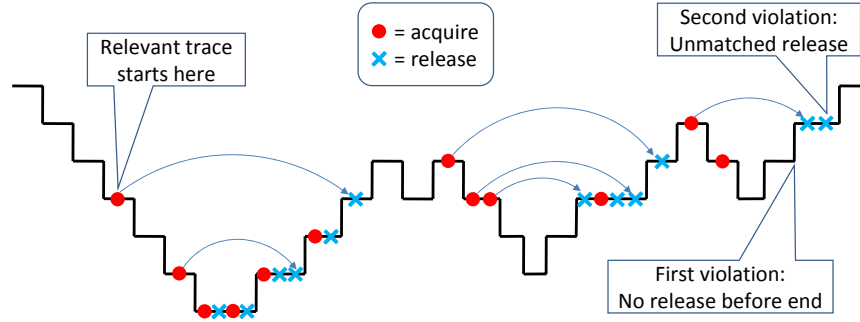


Figure 12.1: Example trace for structured acquire and release of locks.

match the lock acquires. One may want to impose an even stronger locking safety policy: the lock releases should match the lock acquires within the boundaries of each method call. This property is vacuously satisfied when locks are acquired and released in a structured manner using synchronized blocks or methods, like in Java 4+, but it may be easily violated when one implements one's own locking mechanism or uses the semaphores available in Java 5. For example, Figure 12.1 shows an execution violating this basic safety policy twice (each deeper level symbolizes a nested method invocation). First, the policy is violated when one returns from the last (nested) method invocation because one does not release the acquired lock. Second, the policy is also violated immediately after the return from the last method invocation because the lock is released twice by its caller, but acquired only once.

Supposing that the system is instrumented to emit events **begin** and **end** when methods of interest are started and terminated, and that the events **acquire** and **release** are triggered when the lock of interest is acquired and released, respectively, then here is an initial, straightforward way to express this safety policy as a context-free grammar:

$$S \rightarrow \epsilon \mid S \text{ begin } S \text{ end} \mid S \text{ acquire } S \text{ release}$$

Because of the production $S \rightarrow \epsilon$, the pattern is able to terminate (ϵ is the empty trace). This pattern will match any trace with **begin** events in balance with **end** events because these two events occur only in the production $S \rightarrow S \text{ begin } S \text{ end}$ ¹, where they are matched. The S at the beginning of

¹ Note that $S \rightarrow a \mid b$ is shorthand for $S \rightarrow a, S \rightarrow b$.

```

perthread SafeLock(Lock l) {
  event acquire before(Lock l) : call(* Lock.acquire()) && target(l) {}
  event release after(Lock l) : call(* Lock.release()) && target(l) {}
  event begin before() : execution(* *.*(..)) && !within(Lock) {}
  event end after() : execution(* *.*(..)) && !within(Lock) {}

  lr_lazy : S -> epsilon | S acquire M release A,
           M -> epsilon | M begin M end | M acquire M release,
           A -> epsilon | A begin | A end

  @fail { System.out.println("Unsafe lock operation found!"); } }

```

Figure 12.2: JavaMOP specification for the safe lock safety property using the CFG plug-in.

the production allows an unbounded number of these balanced groupings in a row, e.g., **begin begin end end begin end** is a valid trace with two balanced groupings in a row. The production $S \rightarrow S \text{ acquire } S \text{ release}$ is similar to that with **begin** and **end** events, allowing balanced groupings of **acquire** and **release** events. Because all the productions have the same recursive symbol, S , it is possible for **begin/end** pairs to nest within **acquire/release** pairs, and vice versa. Thus a valid trace would be **begin acquire begin end release end begin acquire acquire release release end**.

This pattern is simple and works, however, it has a deficiency in that it must monitor every **begin** of every method in a given program, even those which do not perform any thread synchronization. Next, we present a more efficient grammar that ignores **begin** events that happen before the first **acquire** event in a trace ². The next grammar looks complicated, but we must stress that it is only complicated to improve monitoring efficiency.

$$\begin{aligned}
S &\rightarrow \epsilon \mid S \text{ acquire } M \text{ release } A \\
M &\rightarrow \epsilon \mid M \text{ begin } M \text{ end} \mid M \text{ acquire } M \text{ release} \\
A &\rightarrow \epsilon \mid A \text{ begin} \mid A \text{ end}
\end{aligned}$$

Again, the productions begin with recursive references to S to allow for repetition of balanced groupings. This time, however, the S productions only allow for **acquire** and **release**, not **begin** and **end**. This ensures that only **begins** and **ends** occurring after the first **acquire** are monitored. The

²Events that occur before the first event in a valid trace are ignored by the monitoring algorithm. Events which begin a valid trace are known as *creation events* [62].

non-terminal M stands for “matched” sub-traces, i.e., traces in which all the pairs `begin/end` and `acquire/release` are properly matched, and A stands for sequences of (not necessarily matched) `begin` and `end` events. The A productions are necessary because failures would be reported for `end` events at the end of a trace due to the lack of a matching `begin` that occurred before the `acquire` creation event. The A productions also allow for more `begin` events after the last `release` because we do not want method calls after the last `release` to cause the invocation of the failure handler.

Any (finished or unfinished) execution trace that is not a prefix of a word in the language of S in the context-free grammar (CFG) above is an execution that violates the safety policy. The CFG runtime verification technique presented in this paper and implemented as a logic-plugin in JavaMOP is able to monitor safety properties expressed as CFGs like above³. Monitoring-Oriented Programming (MOP) and JavaMOP (the Java implementation of MOP) are discussed in Section 12.3.

Figure 12.2 shows this `SafeLock` property expressed as a JavaMOP specification, using the CFG logic plug-in. The modifier `perthread` tells JavaMOP to consider events from separate threads as separate traces. This is particularly important as we do not wish `begins` and `ends` of separate threads to cause the pattern to fail. `SafeLock` denotes the name of the specification, while the list after `SafeLock` is the list of parameters to the specification (see below). `SafeLock` is parametric in the `Lock` because we do not wish the `releases` and `acquires` of separate `Locks` to interfere. The keyword `event` introduces an event; the event is first given a name, and then its trigger is defined using an AspectJ [164] advice (before the colon) and a pointcut (after the colon). Of particular note, however, is `!within(Lock)`, used so that we do not monitor the `begins` and `ends` of the `acquire` and `release` methods, which would cause the pattern to fail. JavaMOP’s generic approach to parametric specifications is described in [65] and [57]. Because of this generic approach, the logical formalisms in which properties are expressed need not be aware of the parameters; parameters are added automatically and generically by the JavaMOP framework.

The keyword `lr_lazy` introduces the CFG pattern. Three other possible keywords can be used : `lr`, `laln`, and `laln_lazy`. The two lazy keywords mean that when an event is encountered that causes a pattern match failure, the

³Our CFG plug-in actually supports only the LR(1) and LALR(1) languages; when we use the term context-free we actually mean LR(1)/LALR(1), unless explicitly mentioned otherwise.

failure handler is invoked, but the event itself is not kept in the monitor state so that more failures can be found. If the error causing event were kept, as in the non-lazy keywords, each following event would cause an error, regardless of whether it should. The lazy method is how most programming language parsers work, allowing multiple syntax errors to be caught in one parse. `lr` and `laln` determine which table generation algorithm is used (see Section 12.5 for more information on the two table generation algorithms). The first nonterminal in the pattern is assumed to be the start symbol of the grammar (see Section 12.5.1). Lastly, `@fail` introduces a *pattern failure handler*. The code within the braces following `@fail` runs whenever the pattern fails to match because an invalid event for a given point in a trace is seen. As an alternative, JavaMOP allows `@match` handlers. This gives extra power to our CFG plug-in because context-free languages are not closed under complementation.

The code generated automatically from the JavaMOP specification in Figure 12.2, following the technique described in the rest of the paper, has more than 700 lines of (human unreadable) AspectJ code. We ran this property against a hand-crafted program, which generated the sequence of events seen in Figure 12.1. Both pattern failures were successfully caught in a single run because the CFG plug-in does not add failure inducing events to the monitor state when `lr_lazy` is used. If the keyword `lr` is used instead of `lr_lazy`, only the first failure is caught by the generated monitor.

12.1.2 Contributions

Several approaches have been proposed to monitor context-free properties. For example, Program Query Language (PQL) [192] is based on a description language that encompasses the intersection of context-free languages. Hawk/Eagle [83] uses a fix-point logic and RuleR [31] uses a rule based logic that can specify context-free properties. These approaches propose what we feel are rather complex solutions for monitoring parametric context-free patterns. They generate inefficient monitoring code in many cases, thus preventing practical parametric context-free property monitoring with these systems. The inefficiency of PQL in comparison to JavaMOP with context-free patterns is discussed in Section 12.6. This paper shows that monitoring (the LR(1) and LALR(1) subsets of) parametric context-free patterns is *practical*. We generate *non-parametric* monitors instead of parsers for the defined context-free pattern. Parameters are handled separately, using the algorithm in [65, 57]. This way, we provide an efficient system for monitoring

parametric context-free properties. Our algorithm is totally different from the monitoring algorithm used by the PQL system [192], which mixes the handling of parameters and monitoring of context-free patterns.

When monitoring pattern languages, such as extended regular expressions (ERE) or CFG, we wish to report a match anytime a trace at a given point in program execution matches the pattern. For example, if we have a pattern that is looking for writes to a closed file, we might use the ERE `close write write*`. We wish to report a match on every write, so that we can locate all of the trouble spots in the program. We call this matching every good prefix of the trace because `close write` is a prefix of `close write write` which is a prefix of `close write write write`, and we wish for a match to be reported on each of these prefix traces. We provide two methods to deal with the problem of monitoring good prefixes. One is to modify the LR(1) parsing algorithm with a *stack copying* process. The second method, called *guaranteed acceptance*, was discovered after our work in [195].

Additionally, we extended JavaMOP with *suffix matching*. Suffix matching is, informally, matching against every suffix of a given trace, and is the mode of matching used in Tracematches [25]. A definition of suffix matching can be found in Section 12.4. We also describe optimizations particular to the JavaMOP suffix matching algorithm that improve the efficiency of suffix matching in JavaMOP.

We also performed an extensive evaluation of the CFG monitoring algorithm using the DaCapo [36] benchmark suite and properties used previously to evaluate runtime verification systems [62, 41]. The properties are expressed as CFGs in this evaluation rather than regular expressions for use in JavaMOP. Even when monitored using the CFG plug-in, however, these regular pattern based specifications still use constant space. We thus performed an evaluation of three strictly context-free properties – which use theoretically unbounded space – to show that, even with such properties, the overhead is reasonable, and to show the usefulness of context-free properties. The results of this analysis compare favorably with PQL and Tracematches, two state-of-the-art runtime monitoring systems. One of these properties (`ImprovedLeakingSync`) is expressible in neither PQL nor Tracematches, for reasons explained in Section 12.6. Another of the properties (`SafeFileWriter`), while expressible in PQL, is not expressible in Tracematches because Tracematches has limited ability to express structured properties, rather than the full generality of the LR(1) languages.

Over both the adapted regular properties and the new strictly context-

free properties, the overhead of JavaMOP with CFGs is, on average, over 8 times less than Tracematches on properties that Tracematches is able to express, and over 12 times less than PQL on properties that can be expressed in PQL. On all but 9 of the 45 benchmark/property pairs that generated events⁴, the overhead is less than 5% in JavaMOP with CFGs.

Beyond the work of [195], we have implemented three more versions of the original LR(1)-based cfg plug-in. We now support LALR(1) and a version of both LR(1) and LALR(1) that stay in an error state when an error token is encountered (using `la/lalr` instead of `lr/lalr_lazy`). We discovered and proved the concept of guaranteed acceptance (see Section 12.5.2). We also provide more a comprehensive analysis of the experiments as presented in [195].

12.1.3 Paper Outline

The remainder of the paper is as follows: Section 13.2 illustrates related work; Section 12.3 gives a brief overview of MOP and JavaMOP; Section 12.4 describes suffix matching together with its novel, optimized implementation in JavaMOP; Section 12.5 explains our CFG monitor synthesis technique in JavaMOP, including considerations for suffix matching; Section 12.6 explains our experimental setup and the results of our experiments; Section 19.9 concludes the paper and describes some future work.

12.2 Related Work

12.2.1 Runtime Monitoring

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to [62] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP [58, 55, 62] have their specification formalisms hardwired and *only two of them* share the same logic (LTL). MOP will be discussed in Section 12.3. This observation strengthens our belief underlying MOP — there probably is *no silver-bullet* specification formalism for all purposes. Also, most approaches focus on detecting either violations (pattern failures in CFG) or validations (pattern matches in CFG) of the desired property and support only fixed types of monitors, e.g., online

⁴Overall there are 66 benchmark/property pairs, but 21 of them generate no events, and are removed to more fairly represent the overhead of runtime monitoring.

Approach	Logic	Scope	Mode	Handler
JPaX [126]	LTL	class	offline	violation
TemporalRover [88]	MiTL	class	inline	violation
JavaMaC [166]	PastLTL	class	outline	violation
Hawk [83]	Eagle	global	inline	violation
RuleR [31]	RuleR	global	inline	violation
Tracematches [25]	Reg. Exp.	global	inline	validation
J-Lo [38]	LTL	global	inline	violation
Pal [52]	modified Blast	global	inline	validation
PQL [192]	PQL	global	inline	validation
PTQL [118]	SQL	global	outline	validation

Table 12.1: Runtime Verification Breakdown.

monitors that run together with the monitored program or offline monitors that check the logged execution trace after program termination.

Specifically, there are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces)⁵. The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Table 12.1. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Of the systems mentioned in Table 12.1, only PQL [192], Hawk/Eagle [83], and RuleR [31] can handle arbitrary context-free properties. Hawk/Eagle

⁵Offline implies outline, and inline implies online.

adopts a fix-point logic and uses term rewriting during the monitoring, making it rather inefficient. It also has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available⁶. Because of this and the fact that Hawk/Eagle has not been run on DaCapo [36] with the same properties, we cannot compare our CFG plug-in with Hawk/Eagle. RulerR is a simplification of Eagle that is rule based rather than μ -calculus based, but it still has the ability to specify context-free properties. The current implementation is not built for efficiency or ease of expression with regards to context-free properties. In addition to PQL, we decided to perform comparisons with Tracematches [25], as it is able to monitor a very limited set of context-free properties using compiler-specific support provided by their special AspectJ compiler, ABC [23], and because it is a very efficient system. Pal [52] is able to monitor properties that take calls and returns into account, giving a limited context-free ability for this one case. Pal is implemented for C, rather than Java, and the implementation is not publicly available.

12.2.2 Context-free Grammars in Testing and Verification

Context-free grammars have seen use in several areas of testing and verification not immediately related to runtime monitoring.

Attributed context-free grammars were used as a means to generate test input and output pairs by [91]. The generated test inputs and outputs could be used both the test the specification from which the test grammar was designed, as well as the final implementation of a specification, using automatic test drivers. Their test case generator was capable of generating test cases from the grammar both randomly and systematically. The attributes of the context-free test generation grammars allow a user to attach context sensitive information to parts of the grammar, and allow for refinement of test case generation in order to avoid redundant test cases. Earlier attempts of test case generation via grammars [213, 124, 150] were employed to generate test input only for compilers and parsers rather than programs and their specifications (though, [90] used grammars to generate test cases in much the same way, it could not generate outputs, and it generated far too many

⁶[25] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [83]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [41]. We have confirmed the inefficiency claims of [25] with the authors of Hawk/Eagle.

similar test cases). In [193] context-free grammars were used to generate test data for VLSI (very large scale integration) circuits. [247] applied the concept of test case generation using context-free grammars to Java virtual machine implementations.

All of these approaches differ quite a bit from runtime monitoring. The overhead of these approaches is not nearly so important because they are used to generate test cases in an offline manner, rather than running at the same time as a program that is under testing or a production system, situations for which runtime monitoring is intended. Additionally, runtime monitoring attempts to monitor behavior of a system rather than to generate test cases.

[155] used context-free grammars for an entirely different purpose that is more related to runtime monitoring than is test case generation. They created an interface specification language that uses context-free grammars to provide stub code for model checking. The grammar specifies the sequence of method invocations allowed by the component. The stubs are called by the code under model checking, providing a means of modular model checking. The grammar generated stubs execute during the model checking process to ensure that the non-stub portions of code always follow the specification of method calls given by the grammar. In this way it is similar to runtime monitoring with context-free grammars, as the grammar is used to specify intended behavior, and flag errors when the behavior is not followed at runtime. Our work differs primarily in that it is designed to enforce behavior in a running system rather than to abstract a component, and in that it is parametric, whereas the interface grammars are not.

12.3 MOP Revisited

MOP is an extensible runtime verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an implementation of MOP for the Java programming language. By encapsulating our monitor synthesis algorithm for non-parametric CFG patterns in a JavaMOP logic plug-in, we have achieved an efficient monitoring tool for universally quantified parametric CFG specifications.

Additionally, we have implemented a novel extension of MOP, in JavaMOP, to support suffix matching independent of the language used for pattern specification. We define suffix matching as matching against every suffix of a given event trace, while total matching, also supported by

JavaMOP, attempts to match the entire trace seen at a particular point⁷. Because Tracematches supports *only* suffix matching, and PQL supports a skip semantics more akin to suffix matching than total, we use suffix matching in our experiments.

12.3.1 MOP in a Nutshell

MOP [58, 55, 62] is a formal framework for software development and analysis, in which the developer specifies desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system. MOP is a highly extensible and configurable runtime verification framework. The user is allowed to extend the MOP framework with his/her own logics via *logic plug-ins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture [58], which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently.

In addition to choosing the formalism for a specification, one can also configure the behaviors of the generated monitor through different attributes [55]. Depending upon configuration, the monitors can be separate programs reading events from a log file, from a socket, or from a buffer, or can be inlined within the program at the event observation points; monitors can verify the observed execution trace as a whole or check fragments of the trace. All these configurations are *independent* of the formalism used to specify the property. MOP also provides efficient and logic-independent support for universally quantified parameters [57], which is useful for specifying properties related to more than one object. This extension allows associating parameters with MOP specifications and generating efficient monitoring code from parametric specifications with monitor synthesis algorithms for non-parametric specifications. MOP's generic support for universally quantified patterns simplified our CFG plug-in's implementation.

The JavaMOP implementation provides several interfaces, including a web-based interface, a command-line interface, and an Eclipse-based GUI, providing the developer with different means to manage and process MOP

⁷At a given point in program execution the trace of events seen at that point is evaluated as a complete trace in both total and suffix matching. This means if one is monitoring the pattern e^* , a match must be reported every time the e event occurs.

specifications. JavaMOP follows a client-server architecture to flexibly support these various interfaces, as well as for portability reasons [56]. AspectJ [164] is employed for monitor integration: JavaMOP translates outputs of logic-plug-ins into AspectJ code, which is then merged within the original program by the AspectJ compiler. Five logic-plug-ins are currently provided with JavaMOP: Java Modeling Language (JML) [182], Extended Regular Expressions (ERE), Past-Time and Future-time Linear Temporal Logics (LTL) (see [56] for more details), and Context-Free Grammar (CFG) that is introduced in this paper. Note that these plug-ins can be supported by any implementation of MOP.

One might expect some loss of efficiency for MOP's genericity of logics. However, the JavaMOP-generated monitors yield very reasonable runtime overhead in practice, even for properties requiring intensive runtime checking (on the order of 12% or lower). In most cases it is as efficient as hand optimized monitoring code [62].

12.4 Suffix Matching in JavaMOP

These foundations should be moved in Chapter 4 and/or Chapter 1

According to application requirements, one may want to check a property against either *the whole execution trace* or *every suffix of a trace*. Total matching has been adopted by many runtime verification approaches to detect pattern failures of properties, e.g., JPaX [126] and JavaMaC [166]. Suffix matching has been used mainly by monitoring approaches that aim to find pattern matches of properties, e.g., Tracematches [25]. PQL has a skip semantics, wherein a specification is matched against the trace, but events may be skipped. A precise explanation of PQL's semantics is available in [192]. To define suffix and total matching, we first must define traces and properties:

Definition 1 *Let \mathcal{E} be a set of events. An \mathcal{E} -trace, or simply a trace when \mathcal{E} is understood from context, is any finite sequence of events in \mathcal{E} , that is, an element in \mathcal{E}^* .*

In the context of monitoring, an execution trace is a sequence of events observed up to the current moment, thus execution traces are always finite.

Definition 2 An \mathcal{E} -property P , or simply a property, is a pair of disjoint sets (P_+, P_-) where $P_+ \subseteq \mathcal{E}^*$ and $P_- \subseteq \mathcal{E}^*$; P_+ is the set of pattern matching traces and P_- is its set of pattern failing traces⁸.

Therefore, our notation of property is quite general; for each particular specification formalism, one needs to associate an appropriate property to each formula or pattern in that formalism. For example, for a CFG G , we let $P_G = (P_+, P_-)$ be defined as expected: P_+ is $L(G)$ (the language of G , see Section 12.5.1) and $w \in P_-$ iff w is not the prefix of any $w' \in L(G)$

Definition 3 The total matching semantics of P is a function

$$\llbracket P \rrbracket_{total} : \mathcal{E}^* \rightarrow \{match, fail, ?\}$$

defined as follows for each $w \in \mathcal{E}^*$:

$$\llbracket P \rrbracket_{total}(w) = \begin{cases} match & \text{if } w \in P_+ \\ fail & \text{if } w \in P_- \\ ? & \text{otherwise} \end{cases}$$

The suffix matching semantics of P is a function

$$\llbracket P \rrbracket_{suffix} : \mathcal{E}^* \rightarrow \{match, ?\}$$

defined as follows for each $w \in \mathcal{E}^*$:

$$\llbracket P \rrbracket_{suffix}(w) = \begin{cases} match & \text{if there are } w_1, w_2 \text{ such that } w = w_1 w_2 \text{ and} \\ & \llbracket P \rrbracket_{total}(w_2) = match \\ ? & \text{otherwise} \end{cases}$$

As an example of where suffix matching is useful, consider the `HasNext` property. This property specifies that the Java API method `hasNext` must be called before every call of `next` for an `Iterator`. If we use total matching and a match handler, we must define the pattern as (using a regular expression) “`(hasNext + (hasNext next))* next next`”, to allow for all of the `hasNext` events, or correct uses of `next` that may occur before the two temporally adjacent calls to `next`. If we use suffix matching, because all suffixes of the trace are

⁸More recently we have generalized this concept to generic categories beyond just match and fail [65, 57]. However, match and fail are sufficient for context-free patterns.

tried, the pattern “next next” is sufficient, so long as the `hasNext` event is still defined.

The previous design of JavaMOP supported only total matching. We have implemented a logic-independent extension of JavaMOP to also support suffix matching. This extension is based on the observation that, although total matching and suffix matching have inherently different semantics, it is not difficult to support suffix matching in a total matching setting, if one maintains *a set of monitor states* during monitoring and *creates a new monitor instance at each event* (this amounts to checking the property on each suffix incrementally). However, the situation becomes more complicated when one wants to develop a logic-independent solution, since different logical formalisms can have different state representations. For example, the monitor state can be an integer when the monitor is based on a state machine, a vector like the past-time LTL monitor, or a stack such as the CFG monitor discussed below. Hence, our solution is to *treat every monitor as a blackbox* without assumptions on its internal state. Also, instead of maintaining a set of monitor states in the monitor, we use a wrapper monitor that keeps a set of total matching monitors as its state for suffix matching. For simplicity, from now on, when we say “monitor” without specific constraints, we mean the monitor generated for total matching. When an event is received, the wrapper monitor for suffix matching operates as follows:

1. create a new monitor and add it to the “suffix matching” monitor set;
2. invoke every monitor in the monitor set to handle the received event;
3. if a monitor enters its “pattern fail” state, remove it from the monitor set;
4. if a monitor enters its “pattern match” state, report the pattern match.

The third step is used to keep the “suffix matching” monitor set small by removing unnecessary monitors. Indeed, this implements suffix matching semantics because each total monitor is monitoring a suffix of the current trace, and “pattern match” is only reported if one of the suffixes is valid.

Using our current implementation of suffix matching in JavaMOP, one may further improve the monitoring efficiency if the monitor provides an `equals` method that compares two monitors with regard to their internal states, and a `hashCode` method used to reduce the amount of calls to `equals`. This interface is used to populate a Java `HashSet`: the combination of

the definition of `hashCode` and `equals` ensures the monitors in the `HashSet` are declared duplicates, and removed, based on monitor state rather than memory location. This interface can be easily generated by each JavaMOP logic plug-in because it has full knowledge of the monitor semantics. It is important to note that our approach does *not depend on the underlying specification formalism*.

JavaMOP requires that the logic plug-in designate *creation events* that are the starting events of a validating trace, in order avoid creating unnecessary monitors. A new monitor instance needs to be created only when creation events occur. This feature is especially useful when combined with suffix matching, which otherwise requires creating a new monitor at every event.

12.5 Context-Free Patterns in JavaMOP

We support the LR(1) subset of context-free grammars (CFGs), as well as LALR(1) which is a subset of LR(1). LR(1) is so named because it parses input **L**eft to right and produces a **R**ight-most derivation. The 1 denotes that one token of look-ahead is used. The LA in LALR stands for look-ahead, because, under certain conditions, states in the LR(1) table with different look-aheads may be merged in the LALR(1) table (see Section 12.5.2).

LR(1) can only recognize a subset of the deterministic context-free languages, which are themselves a strict subset of the context-free languages (CFLs) [7, 149]. LR(1), however, is an expressive subset, able to define the syntaxes of most modern programming languages. We chose LR rather than LL because LR recognizes a larger number of grammars without translation. We base our implementation on the Knuth algorithm [168] for LR(1) parser table generation as presented in [7]. While the “action” and “goto” tables generated are normal LR(1) “action” and “goto” tables, the algorithm used to parse has been modified to work in the context of monitoring, explained in detail below. We added a plug-in, which generates LALR(1) using the algorithm presented in [7], because LALR(1) generates smaller tables in some cases. The LALR(1) tables are, at worst, identical to the LR(1) tables; they are never larger. The downside of LALR(1), however, is that it is a strict subset of LR(1). Comparisons between the table size of LR(1) and LALR(1) for the properties we tested can be found in Section 12.6, and an explanation of the LALR(1) optimization can be found in Section 12.5.2. Sections 5.1–5.2.3 cover standard issues related to LR parsing from a monitoring context, while the remainder of Section 12.5 covers new issues specific to adapting

LR parsing to monitoring.

12.5.1 Preliminaries

Some of these should go in Chapter 1

A CFG G is defined as a tuple of the form, $G = (NT, \Sigma, P, S)$. Σ , the alphabet of the CFG, is often referred to as the set of terminals. A very special terminal, $\$,$ represents the *end* of the input. NT is the set of nonterminals. P is the set of productions, which define what strings nonterminals derive. $NT \cup \Sigma$ is often called the set of symbols. Productions have the form $A \rightarrow \gamma$, where $A \in NT$ and γ is a string that either consists of symbols, or is the empty string, ϵ , i.e. $\gamma \in (\Sigma \cup NT)^*$. We use the conventional alternation operator, “ $|$ ”: a production of the form $A \rightarrow \gamma_0 | \gamma_1$ can be equivalently represented as two productions $A \rightarrow \gamma_0$ and $A \rightarrow \gamma_1$. S is the start symbol – that non-terminal from which all strings in the language are derived. For example, $G = (\{A\}, \{a, b\}, P_0, A)$ where $P_0 = \{A \rightarrow aAb | \epsilon\}$ is a simple CFG for the language $\{a^n b^n | n \in \mathbb{N}\}$. The non-terminal A can derive aAb an indeterminate amount of times before deriving ϵ , allowing $a^n b^n$ for any $n \in \mathbb{N}$.

Two important sets are defined for every non-terminal in a grammar: the *first* and *follow* sets. These are used in “action” table construction. The *first* set will be used to decide which terminals in the given grammar define monitor creation events (we shall be more specific about this below). The *follow* set will be useful in illustrating the fundamental challenge of monitoring CFGs. The *first* set of a non-terminal A , denoted $first(A)$, is the set of all terminals t such that the sub-strings which reduce to A may possibly begin with t . The follow set, denoted $follow(A)$, is the set of terminals which follow the strings which reduce to A . These terminals signify a reduction by A .

A reduction is the step whereby a right hand side of a production, γ , is replaced by the left hand side non-terminal in the production. For example, if we have the string $aaabbb$, and we are using our example grammar, G , we can perform a reduction with $\gamma = \epsilon$ resulting in $aaaAbbb$. We can then perform another reduction with $\gamma = aAb$ resulting in $aaAbb$, eventually we reach aAb , which reduces to A (and because A is the start symbol, $aaabbb$ must be in $L(G)$, where $L(G)$ represents the language derived by G).

12.5.2 CFG Monitoring Algorithms

We developed the CFG monitoring algorithms based on existing parsing algorithms. *Action* and *goto* tables are generated from the given CFG pattern and used to advance the monitor at runtime according to the observed events. Moreover, the CFG monitor is unaware of relevant parameters since they are handled by the underlying MOP framework. This greatly simplifies the monitoring algorithm. We next introduce the monitor algorithms in more detail.

CFG Simplification

The CFG plug-in first applies some standard simplifications to the given grammar [149]. The first step of simplification is the removal of non-generating nonterminals (A is non-generating if $\forall s \in \Sigma^*$, s cannot be reduced to A in one or more reductions). The next step in the simplification process is the removal of nonterminals which are unreachable from the start symbol (A is unreachable from the start symbol if there is no string γ that contains A and reduces to the start symbol in any number of steps). The last step removes ϵ -productions from the grammar. After ϵ -productions have been removed from G , resulting in G' , $L(G') = L(G) - \epsilon$.⁹ A monitor matching the empty event trace has little utility, so we feel this is a fair compromise.

Tables and Monitoring

After simplification, the tables are generated for a deterministic push-down automaton (DPDA), that is, a deterministic finite automaton with a stack, which is to be used as a monitor. The algorithm first adds a production to introduce $\$$. If the start symbol of the original grammar was S , it adds a production $S' \rightarrow S\$$. The next step is generating the *canonical LR(1) (or LALR(1)) collection*. The collection consists of *collection items*; each item represents a state in the automaton. A collection item is a set of productions with a marker, $*$, for the current position in the right hand side, augmented with a look-ahead. For example, a possible collection item for the simple **HasNext** pattern $S \rightarrow next\ next$ is $[\{S' \rightarrow * S \$, \$\}, \{S \rightarrow * next\ next, \$\}]$, another is $[\{S \rightarrow next * next, \$\}]$. In both of these collection items, the look-ahead is $\$$. The collection item is first created for the start production,

⁹ The remaining productions are restructured to account for the removal of ϵ -productions without changing the recognized language, other than as mentioned.

```

1  globals action_table, goto_table
2  initialize stack.push(initial_state)
3  procedure monitor(event, stack)
4      locals state, state', stack', A
5      state  $\leftarrow$  stack.top()
6      while (true) {
7          switch (action_table[state, event].action_type) {
8              case shift :
9                  state'  $\leftarrow$  action_table[state, event].next_state
10                 if (state' = error) {
11                     pattern failure
12                     break while
13                 }
14                 stack.push(state')
15                 if (action_table[state', $].action_type = reduce) {
16                     stack'  $\leftarrow$  stack.copy()
17                     monitor($, stack')
18                 }
19                 break while
20             case reduce :
21                 stack.pop(action_table[state, event].pop)
22                 A  $\leftarrow$  action_table[state, event].non_terminal
23                 state'  $\leftarrow$  stack.top()
24                 stack.push(goto_table[state', A])
25                 break switch
26             case accept :
27                 pattern match
28                 break while
29         }
30     }

```

Figure 12.3: CFG Monitoring Algorithm with Stack Copying.

		Action				Goto	
	\$	acquire	release	begin	end		S
0	shift(error)	shift(14)	shift(error)	shift(16)	shift(error)	0	9
1	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	1	error
2	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	reduce(S, 2)	2	error
3	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	3	error
4	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	4	error
5	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	5	error
6	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	reduce(S, 3)	6	error
7	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	7	error
8	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	reduce(S, 4)	8	error
9	accept	shift(15)	shift(error)	shift(17)	shift(error)	9	error
10	shift(error)	shift(15)	shift(3)	shift(17)	shift(error)	10	error
11	shift(error)	shift(15)	shift(error)	shift(17)	shift(4)	11	error
12	shift(error)	shift(15)	shift(7)	shift(17)	shift(error)	12	error
13	shift(error)	shift(15)	shift(error)	shift(17)	shift(8)	13	error
14	shift(error)	shift(14)	shift(1)	shift(16)	shift(error)	14	10
15	shift(error)	shift(14)	shift(5)	shift(16)	shift(error)	15	12
16	shift(error)	shift(14)	shift(error)	shift(16)	shift(2)	16	11
17	shift(error)	shift(14)	shift(error)	shift(16)	shift(6)	17	13

Table 12.2: LALR(1) Tables for SafeLock

$S' \rightarrow S\$$. All productions for S are added to the state with $*$ at the beginning of the right hand side, as can be seen in our example collection item. If there is a production for S such that the first symbol is a nonterminal, A , all the productions of A will also be added, with $*$ at the beginning. This process is transitively closed. The next collection items are generated by advancing $*$ in the production, and then taking the transitive closure for any nonterminals immediately following $*$. Once the collection is created, the tables are generated by treating each item as a state. The productions in the item are considered for each alphabet symbol (including $\$$). If the marker appears in front of said terminal, a *shift* action is generated. The algorithm decides which collection item to shift to by looking for the collection item where there is a production with the same right hand side as the production that caused the shift action, but with $*$ advanced one position. For example, the next collection from $[\{S \rightarrow next * next, \$\}]$ would be $[\{S \rightarrow next next *, \$\}]$. *Shift* actions can never be generated for $\$$; the algorithm disallows it. The handling of *shift* actions by the parsing algorithm can be seen below. If, however, there is a production in the item such as $[\{S \rightarrow next next *, \$\}]$, where the marker appears at the end, and the terminal in question is the look-ahead,

a *reduction* action is generated. More explanation of this algorithm can be found in [7].

When a new event arrives, the monitoring algorithm must decide how to modify the stack. The tables are given in a generic intermediate form, which is converted by the Java shell into two Java arrays.

Pseudo-code for our monitoring algorithm is given in Figure 12.3. The significance of lines 15-18 is explained Section 12.5.2. An entry in *action_table* specifies, via the *action_type*, the type of action: *shift*, *reduce*, or *accept*. Each type of action also requires additional information in order for the algorithm to process said action. An entry in *goto_table* simply identifies the next state for the DPDA. Table 12.2 shows a parse table as it would be used by the algorithm. This is the LALR(1) table for the first SafeLock grammar given in Section 12.1. We show the LALR(1) table because the LR(1) table has 50 states, and the tables can be used interchangeably by the algorithm.

The *shift* action entry contains the next state for the DPDA in the *next_state* field (in parentheses in the shift actions in Table 12.2). A *shift* action simply pushes the next state on the stack, if the next state is *not* the error state (lines 10-14). If, however, the table indicates that the next state *is* the error state, the algorithm reports a pattern fail and breaks *without* touching the stack (lines 11-12). This allows the algorithm to continue to find more pattern failures. After a successful *shift* action, the while loop is broken, allowing execution of the monitored program to continue until the next relevant event (line 19).

The *reduce* action is more complicated. The field *non_terminal* describes which non-terminal (A) the production $A \rightarrow \gamma$ reduces to (the first field in the reduce actions Table 12.2), while the field *pop* denotes how many states to pop from the stack ($|\gamma|$) (the second field in the reduce actions in Table 12.2). The reduction proceeds by popping the specified number of states from the stack and consulting *goto_table* to decide the next state (lines 20-25). The state used for indexing *goto_table* is not the current state, but rather the state at the top of the stack after the specified number of states has been popped (line 21). An indeterminate number of reductions can happen in a row, but there *must* be *shift* at the end of the reduction sequence before the algorithm can terminate for a given event. The reductions happen before the *shift* to simulate the look-ahead of one token specified by the 1 in LR(1).

The *accept* action, which directs the DPDA to signal a pattern match, has no special fields, as no more information is necessary (lines 26-28).

The LALR(1) Optimization

LALR(1) table generation is a standard modification of the LR(1) table generation algorithm [7]. LALR(1) tables are constructed the same way as LR(1) tables, save that states corresponding to collection items with the same *core* are merged as they are discovered. The core of a collection item is that item with the look-aheads removed. This means that LALR(1) tables can be no larger than LR(1) tables, but may be considerably smaller.

However, this process may result in reduce-reduce conflicts (states where a reduction to two or more different nonterminals with the same look-ahead may occur) that do not exist when the LR(1) construction method is used. Shift-reduce conflicts (states where a shift with a given token, a , and a reduction with a as the look-ahead are possible) cannot be introduced because they require that there be two productions in a given collection item such that one has the position marker in front of a terminal t (meaning that t should be shifted), while another production has the marker at the end of the production, with t as the look-ahead (meaning that we should reduce to the left hand side when t is seen). Obviously the conflict must occur *before* merging, because if the shift inducing production were in a state s_0 , while the reduce inducing production were in s_1 , s_0 and s_1 would have different cores, and not be merged. To see how reduce-reduce actions may be introduced, consider the two collection items: $[\{A \rightarrow a, \$\}, \{B \rightarrow a, a\}]$ and $[\{A \rightarrow a, a\}, \{B \rightarrow a, \$\}]$. Before merger, no conflict exists. Looking at the first collection item, there is no conflict between the two productions because it says to reduce to A only when the look-ahead is $\$$, and B only when the look-ahead is a . The collection item after merger, however, is: $[\{A \rightarrow a, \$\}, \{B \rightarrow a, a\}, \{A \rightarrow a, a\}, \{B \rightarrow a, \$\}]$, which contains two reduce-reduce conflicts (one on look-ahead a , the other on $\$$). The only way to check if such a conflict is introduced by the LALR(1) merger is to generate LR(1) tables to see if there is still a reduce-reduce conflict.

Handling the End of Trace

The (LA)LR(1) algorithm assumes that the string of terminals to be evaluated is *completely known ahead of time*. Thus, it knows where the end of the string (denoted as $\$$) is. This is important because some reductions happen with the $\$$ symbol as the look-ahead, and the accept action can *only* be recognized when the next input is $\$$. To be consistent with our notion of monitoring, it must be possible to consider the trace prefix at a given point

in a run of a program as an *entire* trace. The algorithm must then assume \$ after every event.

Our implementation of the algorithm attempts to reduce with \$ as the look-ahead after *every* valid *shift* (lines 15-18). The problem with reducing with \$ as the look-ahead where possible is that all state of the current trace evaluation is lost. This means that the monitor could only accept the minimal trace that matches the CFG pattern if no special care were taken.

Since our notion of monitoring reports pattern matches for every current trace that matches the pattern¹⁰, one possible option is to *copy* the stack before we perform any reductions with \$ as the look-ahead.

This copying ensures that the stack is intact for the next, and subsequent events, allowing for multiple pattern matches. For example, consider the language denoted by the regular expression ab^* . While we would suggest using the ERE plug-in for such a language, it is a clear example to illustrate the effect of copying. With no copying the algorithm would accept for only a . Because it popped during the pattern match phase, if it sees a b it will report failure. With copying it will report success for a , and then success again on the input of b , and for any subsequent input of b . An important optimization to copying is to only copy the stack if there is a reduction with \$ as the look-ahead, rather than blindly for every *shift* operation. This optimization will not help ab^* , but it will help for many other languages. In fact, for $\{a^n b^n | n \in \mathbb{N}\}$, only one copy is necessary no matter how long the input. Any grammar accepting unbounded repetition at the end of the pattern (like ab^*), will require copying on each input of the repeated character.

Our experience with stack copying led us to an important observation: when there is a reduction with \$ as the look-ahead, acceptance is *guaranteed*. That is to say, there is never a situation in which there is a reduction with \$ as the look-ahead that results in a parse failure. This is a consequence of the parse table generation algorithm, and Section 12.5.4 covers the correctness of this notion, which we refer to as *guaranteed acceptance*. Using guaranteed acceptance to accept whenever a reduction with \$ as the look-ahead is possible is another alternative for matching all good prefixes of a pattern. Guaranteed acceptance is always correct. We maintain the discussion on stack copying because the proof of the stack copying algorithm is easier to understand, and because it is an interesting, though less practical, alternative to guaranteed acceptance.

¹⁰This is irrespective of suffix matching which actually generates multiple monitors.

12.5.3 CFG-plugin Implementation

The CFG plug-in allows the user to specify a number of events and a CFG specifying allowable event traces. The events become the terminals of the CFG, i.e., Σ . The translation steps from specification to working Java code gradually transform the specification into AspectJ join points (the events) and aspects (the synthesized monitors), which are then woven into the original application using any off-the-shelf AspectJ compiler.

Suffix Matching with CFGs

As described in Section 12.4, several features are needed for monitors to support optimized suffix matching.

The first is identification of monitor creation events¹¹. As already mentioned, monitor creation events are events which, when encountered as the *first* event in a trace, would not lead to an immediate failure. For CFGs this would imply an event that can begin a sub-string which reduces to the start symbol. This is the same as the definition of *first* set as given earlier. Thus, the monitor creation events for the CFG plug-in are those events which are in $first(S)$, where S is the start symbol for the grammar.

Additionally, it is necessary to define a hash encoding for CFG based monitors because our suffix matching algorithm uses a `HashSet` to find monitors with *potentially* equivalent states quickly. We decided that two simple defining aspects of CFG based monitors are stack depth and the current state of the monitor (the top of the stack). We chose to xor them together (a broadly used operation for combining two binary quantities into one quantity representative of the two in the same number of bits) because the `hashCode` method must return an integer. Lastly, we need an equality method (to resolve collisions) defining when two CFG based monitors have *actually* equivalent states. Two CFG monitors can only be equal iff they have the same stack contents. It will be fairly rare for two proper CFG monitors to be equivalent, as they do not have finite state like the other logic plug-ins of MOP. Thus, it is important for failed equality testing to be fast. Because of this, we check to see if two monitors have the same stack depth before beginning element-wise comparisons.

¹¹Though, as mentioned, this is also necessary for total matching.

12.5.4 Proofs of Correctness

We next prove the correctness of the proposed CFG monitoring algorithms.

First, we prove the online monitoring algorithm for CFG using stack copying correct. We achieve this by showing that our algorithm detects pattern failures and pattern matches of the observed trace in the same way as the ASU parsing algorithm[7], as given in Figure 12.4.

Theorem: *For every finite prefix of a (possibly infinite) program trace¹² and a CFG pattern, the MOP algorithm will notify a failure of the pattern if the ASU algorithm would notify a parse failure due to a bad token, and pattern match if ASU would notify success, given that prefix as total input.*

Proof: First, we construct a new parsing algorithm, as shown in Figure 12.5. This new algorithm can be proved equivalent to the one in Figure 12.4 as follows. The major difference between these two algorithms is that the pointer (*ip*) increment is moved to the outer loop in Figure 12.5. This change does not affect the behavior of the algorithm:

1. For a shift action, both algorithms carry out the same operation except that Figure 12.4 increases the pointer and continues to the next action, while Figure 12.5 breaks the inner loop, increases the pointer in the outer loop, and then continues to the next action. Both are equivalent.
2. For reduction, Figure 12.5 chooses to stay in the inner loop, which is identical to Figure 12.4, and continues the loop without increasing the pointer.
3. For acceptance (pattern match in monitors), both algorithms are identical.

Now we can prove the correctness of the monitoring algorithm in Figure 12.3 by comparing it with the modified parsing algorithm in Figure 12.5.

The major difference distinguishing the monitoring algorithm from ASU is that the former has to wait for the next event extracted from the execution of the monitored program while the latter can actively retrieve the next token, which is handled in the outer loop in Figure 12.5. Therefore, we only need to prove that the *monitor* procedure in Figure 12.3 produces the same result as the inner loop in Figure 12.5, given the same state and event to process.

¹²Each prefix is an \mathcal{E} -trace at a given point in a program as per Definition 1.

```

1  globals stack, ip, action_table, goto_table
2  initialize stack.push(initial_state), ip ← 0
3  procedure parse()
4    locals state, state', a, A
5    while (true) {
6      state ← stack.top()
7      a ← get_token_at(ip)
8      switch (at[state, a].action_type) {
9        case shift :
10         state' ← action_table[state, a].next_state
11         if (state' = error) {
12           report_error
13           advance ip
14           continue while
15         }
16         stack.push(state')
17         advance ip
18         continue while
19        case reduce :
20         stack.pop(action_table[state, a].pop)
21         A ← action_table[state, a].non_nonterminal
22         state' ← stack.top()
23         stack.push(goto_table[state', A])
24         continue while
25        case accept :
26         accept
27         return
28      }
29    }

```

Figure 12.4: ASU Algorithm.

```

1  globals stack, ip, action_table, goto_table
2  initialize stack.push(initial_state), ip  $\leftarrow$  0
3  procedure parse()
4      locals state, state', a, A
5      while (true) {
6          a  $\leftarrow$  get_token_at(ip)
7          while (true) {
8              state  $\leftarrow$  stack.top()
9              switch (at[state, a].action_type) {
10                 case shift :
11                     state'  $\leftarrow$  action_table[state, a].next_state
12                     if (state' = error) {
13                         report_error
14                         break while
15                     }
16                     stack.push(state')
17                     break while
18                 case reduce :
19                     stack.pop(action_table[state, a].pop)
20                     A  $\leftarrow$  action_table[state, a].non_nonterminal
21                     state'  $\leftarrow$  stack.top()
22                     stack.push(goto_table[state', A])
23                     continue while
24                 case accept :
25                     accept
26                     return
27             }
28         }
29         advance ip
30     }

```

Figure 12.5: Modified ASU Algorithm.

It is straightforward to compare both pieces of code: the only difference between them is the *stack copying* (lines 14-17) in Figure 12.3. It is needed because we wish to continue parsing *after an accept*, and because we can never actually see \$ as an event. We copy the stack after a shift and check for actions with \$ as the input. The only actions possible on this recursive call are reduce and accept because \$ can never be shifted¹³. Due to this, the recursion is always bounded at depth one. This is the major difference between the MOP and ASU algorithms. Because \$ can never be an event, we must speculatively guess the end of input after every symbol seen. The recursive call must happen iff there is a valid reduction with \$ as the look-ahead. Because the algorithm repeats until a shift action, error, or accept happens, we ensure that, if the recursive call happens, it must happen after the processing of each input¹⁴. Cloning the stack allows us to reduce and accept, while still maintaining the original stack to continue monitoring events as if the end of input had *not* been seen. Thus, this change is equivalent to the ASU algorithm in terms of language recognition because both possibilities (the arrival or non-arrival of \$) are explored. That is, the MOP algorithm will report accept for a given prefix if ASU would, given that prefix as its total input, and it, additionally, retains enough state to continue parsing future (longer) traces. Violation is handled identically in both algorithms. \square

We next prove the correctness of our online monitoring algorithm for CFG using the notion of *guaranteed acceptance*. It is achieved by showing that a reduction with \$ as the look-ahead must result in accept with \$ as look-ahead¹⁵ in one or more reductions. Figure 12.6 shows the algorithm modified to take advantage of guaranteed acceptance. Note that there is no longer an `accept` case because accept is discovered in the `shift` case, on lines 15-16.

Theorem: *If the action table entry for a given state specifies reduction with \$ as look-ahead, the stack copying processes must lead to an acceptance. That is to say, in one or more reductions with \$ as the look-ahead, an accept action must occur with \$ as the look-ahead.*

Proof: From the canonical LR(1) construction algorithm [7], we know that a given non-terminal B can only be reduced to with look-ahead terminal

¹³This is a property of the CFG parsing table generation algorithm, which we use without proof. It is obvious, however, because \$ is not a part of the original grammar.

¹⁴Accept need not be considered because it can only happen when the input is \$, which only occurs during a recursive call.

¹⁵Note that this is the only look-ahead ever possible for accept.

```

1  globals action_table, goto_table
2  initialize stack.push(initial_state)
3  procedure monitor(event, stack)
4      locals state, state', stack', A
5      state  $\leftarrow$  stack.top()
6      while (true) {
7          switch (action_table[state, event].action_type) {
8              case shift :
9                  state'  $\leftarrow$  action_table[state, event].next_state
10                 if (state' = error) {
11                     pattern failure
12                     break while
13                 }
14                 stack.push(state')
15                 if (action_table[state', $].action_type = reduce) {
16                     pattern match
17                 }
18                 break while
19             case reduce :
20                 stack.pop(action_table[state, event].pop)
21                 A  $\leftarrow$  action_table[state, event].non_terminal
22                 state'  $\leftarrow$  stack.top()
23                 stack.push(goto_table[state', A])
24                 break switch
25         }

```

Figure 12.6: CFG Monitoring Algorithm with Guaranteed Acceptance.

a , if there exists some production $C \rightarrow \gamma_0 B a \gamma_1$ or a sequence of productions $C \rightarrow \gamma_0 B_0 a \gamma_1$, $B_0 \rightarrow \gamma_2 B_1$, $B_1 \rightarrow \gamma_3 B_2$, ... $B_n \rightarrow \gamma_{n+2} B$. Note that the sequence may contain cycles of one or more production, such as a production $C \rightarrow bC$, but there must be a finite amount of “cycle reductions” because there is a finite amount of input, and the CFG simplification we perform removes non-generating nonterminals¹⁶.

In the algorithm, $\$$ is treated as a special terminal that exists in only one production, the start production $S' \rightarrow S\$$, where S is the original start symbol of the grammar. This production is added by the algorithm and is the only existence of $\$$ in the grammar. Intuitively, when the contents of the parse stack *correspond* to S , the algorithm accepts. This means that if we can reach a stack containing only state corresponding to S through one or more reductions with $\$$ as the look-ahead, we can accept. The original ASU algorithm and our version of the algorithm with stack copying perform all of these reductions.

As a result of the two facts above, a reduction with $\$$ as look-ahead, for non-terminal A , can only occur in the table if there is some sequence of productions $S \rightarrow A_0 \$$, $A_0 \rightarrow \gamma_0 A_1$, $A_1 \rightarrow \gamma_1 A_2$, ... $A_n \rightarrow \gamma_n A$, or the production $S \rightarrow A\$$. If we have $S \rightarrow A\$$, then we can obviously accept on the reduction to A because this will result in a stack with only contents corresponding to A ($A = S$ from above), which is the accept condition, or there will be some finite number of cycles with non-terminal A that eventually leads to a stack containing only contents corresponding to A because there must be a finite amount of cycle reductions. If, however, we have the sequence of productions, we can still accept because there is a sequence of reductions from A to $S \rightarrow A_0 \$$ given by the sequence of productions. Again, even if the sequence contains cycles, eventually acceptance must be reached because there must be a finite amount of cycle reductions. \square

12.6 Evaluation

We evaluated the JavaMOP CFG plug-in and compared its performance to PQL and Tracematches on the DaCapo benchmark suite [36]. We used the LR(1) tables, with the lazy algorithm, and suffix matching. We feel this gives a worst case. LR(1) tables are of greater or equal size, so they cannot be faster than the LALR(1) tables. The lazy mode has the potential to be

¹⁶It is clear that only non-generating nonterminals could have infinite cycles, because the table generation algorithm works bottom up, and chooses shift over reduce on conflict.

slower because it continues to modify the stack after a failure, while the normal mode does not. Suffix matching is obviously slower than total trace matching because it creates one total match monitor for every suffix of the trace.

12.6.1 Experimental Settings

Our experiments were carried out on a machine with 1.5GB RAM and Pentium 4 2.66GHz processor. The operating system used was Ubuntu Linux 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs [36]: `antlr`, `bloat`, `chart`, `eclipse`, `fop`, `hsqldb`, `jython`, `luindex`, `lusearch`, `pmd`, and `xalan`. The provided default input was used together with the `-converge` option to execute the benchmark multiple times until the execution time falls within a coefficient of variation of 3%. The average execution time of six iterations after convergence are then used to compute the runtime overhead. Therefore, Table 12.3 percentages should be read “ ± 3 ” (meaning negative numbers are possible).

12.6.2 Properties

The following general properties borrowed from [41] were checked in the evaluation:

- **HashMap**: An object’s hash code should not be changed when the object is a key in a **HashMap**;
- **HasNext**: For a given iterator, the `hasNext()` method should be called between all calls to `next()`;
- **Safeliterator**: Do not update a **Collection** when using the **Iterator** interface to iterate its elements.

We also defined three new properties to showcase the power of the CFG plug-in; they are all properly context-free:

- **ImprovedLeakingSync**: The original `LeakingSync` specified in [41] *only* allows synchronized accesses to synchronized collections. This causes spurious failures because the synchronized methods call the unsynchronized versions. Our improved version allows calls to the unsynchronized methods so long as they happen within synchronized calls.

- **SafeFileInputStream:** `SafeFileInputStream` is a modification of our `SafeLock` property from Figure 12.2. It ensures that a `FileInputStream` is closed in the same method in which it is created.
- **SafeFileWriter:** `SafeFileWriter` ensures that all writes to a `FileWriter` happen between creation and close of the `FileWriter`, and that the creation and close events are matched pairs.

More properties have been checked in our experiments; we chose the first three regular-language-based properties (`HashMap`, `HasNext`, and `SafeIterator`) to include in this paper because they generate a comparatively larger runtime overhead. We excluded those with little overhead in JavaMOP. For every property, we provide overhead percentages for JavaMOP, as well as PQL and Tracematches where possible. We run the JavaMOP monitors in suffix matching mode; the decentralized indexing of monitors was used in all the experiments (see [62]). We chose the AspectJ compiler 1.5.3 (AJC) in the evaluation to compile the JavaMOP generated monitoring AspectJ code. Guaranteed acceptance and stack copying have the same performance on each of these patterns because none of the properties is able to generate more than one pattern match from a given parameter instance, meaning that the number of stack copies is very minimal. Additionally, the properties have been written in a method that ensures minimal stack size (such as using left recursion instead of right recursion). For Tracematches we used the most recent published version from [37].

12.6.3 Results

Table 12.3 shows the percent overheads of JavaMOP using the CFG plugin, PQL, and Tracematches. N/E refers to specifications that were not expressible. Negative numbers can be attributed both to the 3% noise in the measurements and instruction cache layout changes due to the weaving process. Tracematches is unable to support `ImprovedLeakingSync` because the property is truly context-free. PQL is also unable to support it because it requires events corresponding to the beginning and end of synchronized method calls, and PQL can only trigger events on the end of method calls. Tracematches cannot support `SafeFileWriter` because it is a pure context-free specification. However, Tracematches *can* support `SafeFileInputStream` because it has the ability to access call stack depth via the `cflowdepth` pointcut term, which is provided only by the ABC compiler for AspectJ.

	HashMap			HasNext			SafeIterator		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	3	6	0	1	2	3	2	82	0
bloat	14	9	-2	1112	5929	2452	627	8694	11258
chart	-1	1	-1	-1	3	0	2	50	11
eclipse	0	1	1	0	2	-1	-2	1	2
fop	3	2	0	0	2	-1	-1	24	5
hsqldb	0	3	15	0	6	15	0	78	17
jython	0	23	15	0	0	13	0	12	16
luindex	1	8	1	-2	93	2	3	181	9
lusearch	1	1	8	-1	59	9	4	132	34
pmd	-1	0	3	191	1870	52	178	1334	175
xalan	0	5	1	0	0	2	1	53	10
	ImprovedLeakingSync			SafeFileInputStream			SafeFileWriter		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	1	N/E	N/E	3	113	-1	2	22	N/E
bloat	13	N/E	N/E	1	128	0	27	97	N/E
chart	4	N/E	N/E	0	29	1	0	37	N/E
eclipse	1	N/E	N/E	-2	3	0	-2	1	N/E
fop	1	N/E	N/E	-2	58	-1	-2	47	N/E
hsqldb	1	N/E	N/E	1	280	21	2	95	N/E
jython	41	N/E	N/E	0	937	12	1	crashes	N/E
luindex	1	N/E	N/E	-1	233	6	0	33	N/E
lusearch	2	N/E	N/E	-1	137	7	0	49	N/E
pmd	36	N/E	N/E	-1	547	1	-2	658	N/E
xalan	3	N/E	N/E	-1	90	3	-2	164	N/E

Table 12.3: Average percent runtime overhead for JavaMOP CFG (MOP), PQL, and Tracematches (TM) (convergence within 3%); N/E means “not expressible”.

Property	HashMap	HasNext	Safeliterator	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
antlr	0	0	1990	8472	0	0
bloat	361519	143103032	75944328	5587905	259	385
chart	8773	6819	569345	634260	0	0
eclipse	20888	3252	32759	74630	930	0
fop	17265	281	49959	182407	12	0
hsqldb	0	0	0	0	0	0
jython	443	106	177554	23969673	544	0
luindex	9615	28140	82162	1559386	1114	0
lusearch	416	0	405428	1291992	0	0
pmd	11354	33294563	25476563	26291289	10	32
xalan	124155	0	1009649	5146036	13604	0

Table 12.4: Number of events handled by JavaMOP

Over one run of the entire DaCapo benchmark suite, more than 355 *million* events (Table 12.4) were triggered. Tracematches has the same number of events throughout the tests because it uses the same instrumentation technique as JavaMOP. We had no good method to obtain the number of events generated in PQL; we assume it was less because PQL performs a static optimization which removes unnecessary optimization points. It is interesting to note that in the cases of `HasNext` with `antlr`, `lusearch`, and `xalan` that there are no events, despite the fact that these three benchmarks have events for `Safeliterator`. The reason for this is that the `Safeliterator` instruments `Collection.remove`, so it is possible for `Safeliterator` to have events in programs with no actual `Iterators`.

The average overhead of JavaMOP over 45 program/property pairs that actually generate events is 50%. There are two considerations here, however: (1) we chose specifically those properties that generated the largest overheads (`HasNext` and `Safeliterator` in `bloat`), (2) when the two largest overheads are removed, the average over the remaining 43 pairs drops to a very reasonable 12%. Further, the average JavaMOP overhead for properties expressible in PQL that generated events was 61% over 36 pairs, while PQL’s overhead on these same properties was 583%. Similarly, for Tracematches expressible properties that generated events, JavaMOP’s overhead was 64% over 33 pairs, while Tracematches was 414%. Tracematches, PQL, and JavaMOP all feature the same two pairs which have extremely large overhead compared to the median (`HasNext` and `Safeliterator` in `bloat`). When these two pairs are removed from the three averages, the average overhead for JavaMOP with respect to PQL expressible properties is 12%, while PQL still weighs in at 199%. Tracematches is comparable to JavaMOP, with JavaMOP and

Property	Original	HashMap	HasNext	Safeliterator
antlr	2.3 / 10.1	2.0 / 10.6	1.8 / 10.6	2.0 / 10.8
bloat	5.6 / 8.9	6.9 / 8.9	5.9 / 8.7	541.0 / 10.6
chart	20.1 / 11.3	20.8 / 11.4	17.0 / 11.3	20.7 / 11.5
eclipse	27.0 / 22.1	30.7 / 22.2	27.4 / 22.1	28.6 / 22.3
fop	12.3 / 9.1	13.2 / 9.2	10.9 / 9.0	10.2 / 9.1
hsqldb	80.8 / 7.6	80.2 / 7.6	76.4 / 7.5	77.5 / 7.6
jython	3.9 / 19.0	4.1 / 19.0	3.8 / 19.0	3.9 / 19.1
luindex	4.2 / 6.9	4.0 / 7.0	4.6 / 6.9	4.7 / 7.1
lusearch	5.2 / 6.2	5.2 / 6.3	5.7 / 6.2	5.3 / 6.3
pmd	22.0 / 8.6	22.3 / 8.7	24.0 / 8.6	888.1 / 8.9
xalan	21.7 / 10.2	23.8 / 10.5	26.2 / 10.2	29.1 / 10.3
Property	Original	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
antlr	2.3 / 10.1	2.1 / 10.7	2.4 / 10.7	2.2 / 10.7
bloat	5.6 / 8.9	7.9 / 10.0	5.0 / 8.9	5.6 / 8.9
chart	20.1 / 11.3	17.0 / 11.3	17.8 / 11.3	16.4 / 11.3
eclipse	27.0 / 22.1	28.9 / 22.1	30.7 / 22.1	27.1 / 22.1
fop	12.3 / 9.1	14.6 / 9.2	11.9 / 9.0	12.0 / 9.0
hsqldb	80.8 / 7.6	87.2 / 7.5	78.2 / 7.5	79.3 / 7.5
jython	3.9 / 19.0	4.0 / 19.2	4.0 / 19.1	3.6 / 19.1
luindex	4.2 / 6.9	5.6 / 7.0	4.2 / 6.9	4.6 / 6.9
lusearch	5.2 / 6.2	5.8 / 6.4	5.6 / 6.3	5.7 / 6.3
pmd	22.0 / 8.6	22.2 / 8.8	24.2 / 8.6	22.9 / 8.6
xalan	21.7 / 10.2	24.4 / 10.3	22.0 / 10.3	26.5 / 10.2

Table 12.5: Maximum memory usage in MB (Maximum Heap Memory Usage) / (Maximum Non-Heap Memory Usage).

Tracematches both at 12%. Since Tracematches does not support the full generality of (deterministic) context-free grammars, we view comparable performance to Tracematches as favorable to our approach, especially given that, in the overall data set, our average overhead is over 8 times lower than Tracematches’ overhead.

The largest overheads seen, across all three systems, are for **Safeliterator** and **HasNext** in **bloat**. This is due to **bloat**’s extensive use of iterators. **Bloat** is a bytecode optimizer, which uses iterators to process bytecode. **PQL** and **Tracematches** perform worse on **Safeliterator** than they do on **HasNext**, while our performance is the opposite. The reason for this is that **HasNext** creates

a far larger number of monitors in JavaMOP because it creates a monitor for every call to `next`, while `Safeliterator` only creates monitors on a call to `create`. The pattern for `Safeliterator`, however, is more complex. This shows that JavaMOP has, relatively speaking, more overhead in generating and handling the monitor set for suffix matching than it does in matching the pattern, while PQL and Tracematches overheads are more affected by the complexity of the pattern. Note that JavaMOP with CFGs far outperforms both PQL and Tracematches on these 2 program/property pairs.

`SafeFileInputStream` is an interesting case: it is required to match the begin and end of methods. Instrumenting the begin and end of *every* method would be atrociously slow, however. We perform a static analysis which finds those methods in which `FileInputStream`'s are actually used. Then, we instrument only those methods for begin and end. Because Tracematches, also, is pointcut based, we are able to perform the same optimization for Tracematches, so the numbers shown are with the optimization enabled. PQL is not pointcut based so the optimization cannot be applied; however, the PQL property does not match begins and ends of methods (recall: PQL can only match the ends), so this is not an issue. In PQL we specify `SafeFileInputStream` by using an interesting PQL-specific feature called `within`. The idea of `within` is that a property matches only *within* a given method or methods matching a particular pattern (in the case of `SafeFileInputStream` we use the pattern `...` which specifies all methods of all classes). Additionally, PQL will only instrument the same methods that JavaMOP and Tracematches instrument because `within` only instruments methods which can generate relevant events.

The memory overhead is reasonable in our experiments: overall, it is 33% on average with a 4% median (see Table 12.5 for a pair-wise breakdown). There are two extreme cases of memory overhead caused by JavaMOP monitors: `bloat-Safeliterator` and `pmd-Safeliterator`. Our investigation shows that both programs, `bloat` and `pmd`, make intensive use of vectors, and create numerous iterators to do computation over the vectors throughout the execution. Note that every creation of the iterator leads to the creation of a monitor instance for `Safeliterator` using our technique. Hence, a huge number of monitor instances were created in these two benchmarks. While the iterator object is usually used in a small scope and then released, the vectors are not released until the end of the execution, preventing the removal of the created monitor instances. In other words, all the monitor instances created during the execution of `bloat` and `pmd` were kept alive until the execution

	HashMap	HasNext	Safeliterator	ImprovedLeakingSync	SafeFileInputStream	SafeFileWriter
LR(1) states	6	5	15	19	20	18
LALR(1) states	6	5	15	15	8	11

Table 12.6: Comparison of LR(1) and LALR(1) tables.

ended, resulting in the observed massive memory usage. On the contrary, we can see that a large number of monitors were also created for **bloat-HasNext** and **pmd-HasNext** but with much less memory overhead. **HasNext** has one monitor created for every iterator object, and when the iterator is released, the corresponding monitor will also be removed. Since most iterators were released shortly after creation, only a few monitors existed at the same time during the execution resulting in much lower memory overhead. Compared with the results of Tracematches [25], we believe there is still some room for improvement with regard to memory usage in our approach. The memory overhead of our approach does not cause unnecessary loss of performance during the evaluation, indicating that it is not a bottleneck for the efficiency of monitoring.

Table 12.6 shows a comparison of parse table size between LR(1) and LALR(1) in terms of number of states. No reduce-reduce conflicts were introduced in any of our properties, and the space savings can be significant. Although, in either case, the tables are small enough that the size difference has no effect on runtime. It is interesting to note that the three regular language properties see no savings from LALR(1).

Our experiments with guaranteed acceptance found no benefit to the patterns we tested because, as mentioned, each pattern can only accept once per parameter instance, and the stack depth is kept to a minimum by using careful grammar design. Not all properties, however, can or should be written in such a way that only one acceptance can be generated per parameter instance. In extreme cases, such as patterns that feature unbounded repetition at the end of the pattern, such as a^* , guaranteed acceptance can provide a *lower asymptotic complexity*. Consider the grammar $S \rightarrow \epsilon | aS$ which monitors a^* . For each a that arrives, with guaranteed acceptance a *constant time*¹⁷ step of adding a to the stack and accepting occurs. With stack copying, the stack must be copied on each arrival of a . This copying, however, will take time linear in the amount of times a has been previously seen. Thus, with guaranteed acceptance, the monitoring

¹⁷It is constant except when the stack size must be grown.

time is linear in the number of a events in the program run while stack copying is quadratic.

12.7 Conclusions and Future Work

We implemented a CFG logic plug-in for JavaMOP using a modified LR(1) parsing algorithm. We also implemented an optimization of table generation, which uses the LALR(1) state merging technique, leading to smaller tables, but may result in extra reduce-reduce conflicts. Our first modification to the algorithm is based on the novel idea of *copying* the stack in order to “predict” a possible reduction with $\$$ (end of string) as a look-ahead without destroying the state of the monitor. An important optimization and simplification possibility is *guaranteed acceptance*, wherein the algorithm accepts when a reduction with $\$$ as look-ahead is possible; this saves the copying operation, which can take arbitrarily long to perform, since the stack is unbounded. We showed, empirically, that our algorithm is efficient and faster than the state-of-the-art for monitoring CFG properties. We also extended JavaMOP with suffix matching in order to fairly compare JavaMOP with PQL and Tracematches. Tracematches, however, cannot handle arbitrary context-free patterns.

We have also begun work on a logic called PtCaRet as another specification formalism to support structured specifications. PtCaRet is past time linear temporal logic (PTLTL) with calls and returns. It is a super set of PTLTL that provides operators which apply only on a function/method local level.

Exercises

Exercise 21 Can the *SafeLock* property described in Section 12.1.1 (see Figure 12.2, too) and defined there using the CFG formalism be also defined in PTCARET (Chapter 11)? Why or why not.

Exercise 22 Manually execute the CFG monitoring algorithm in Figure 12.6 for the *SafeLock* property in Section 12.1.1 with its LALR(1) table in Table 12.2, on the following observed trace: *begin begin acquire acquire release release end begin acquire release end end begin end*. Explain only the major steps. The purpose of this exercise is to demonstrate that you understand the CFG monitoring algorithm, including the handling of the $\$$ event.

Exercise 23 *Describe a derivative-based monitoring algorithm for CFG patterns.*

Exercise 24 *Compare monitoring a regular language using monitors for regular expressions (automata or derivative-based) vs. monitors for CFGs specifying the same regular language. Would there be any asymptotic (time or space) penalty to pay for using the CFG monitoring approach?*

Chapter 13

Efficient Monitoring with Deterministic String Rewrite Systems

Abstract: Early efforts in runtime verification show that parametric regular and temporal logic specifications can be monitored efficiently. These approaches, however, have limited expressiveness: their specifications always reduce to monitors with finite state. More recent developments showed that parametric context-free properties can be efficiently monitored with overheads generally lower than 12–15%. While context-free grammars are more expressive than finite-state languages, they still do not allow every computable safety property. This paper presents a monitor synthesis algorithm for string rewriting systems (SRS). SRSs are well known to be Turing complete, allowing for the formal specification of any computable safety property. Earlier attempts at Turing complete monitoring have been relatively inefficient. This paper demonstrates that monitoring parametric SRSs is practical. The presented algorithm uses a modified version of Aho-Corasick string searching for quick pattern matching with an incremental rewriting approach that avoids reexamining parts of the string known to contain no redexes.

13.1 Introduction

Runtime verification (RV) is a formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against its requirements at runtime, as in testing. A large number of runtime verification techniques and systems, including TemporalRover [88], JPaX [126], JavaMaC [166], Hawk/Eagle [83], Tracematches [9, 25], J-Lo [38], PQL [192], PTQL [118], MOP [62, 58], Pal [52], RuleR [31], etc., have been developed recently, and the overall approach has gained enough traction to spawn its own conference [29]. In a runtime verification system, monitoring code is generated from the specified properties and integrated with the system to monitor. Therefore, a runtime verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a means to instrument programs. The chosen specification formalism determines the expressivity of the runtime verification approach and/or system.

Monitoring safety properties is arbitrarily complex [235]. Early developments in runtime verification, showed that *parametric* regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. A parametric monitor associates monitor states with different object instantiations for the given parameters. This allows for specification of properties about the relationships of objects, e.g., a relationship between a Collection object and its associated Iterator objects in Java¹. As shown by experiments with Tracematches [25] and the most recent experiments using JavaMOP [160], parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 15% or lower.

However, both regular expressions and temporal logics are monitored using finite automata, so they have inherently limited expressivity. More specifically, most runtime verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such runtime verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. PQL [192], Hawk/Eagle [83], and RuleR [31] provide more expressive logics, but these are relatively inefficient [9, 25, 62]. More

¹ Typestates [259], a popular concept in software engineering and software analysis, can be monitored with parametric monitors that have only one parameter.

recently, JavaMOP was extended to support efficient context-free monitors with runtime overheads very similar to the earlier finite-state logics [196]. While this work allows for checking many structured properties, it does not have the full power to specify any possible safety property. In this paper, we introduce an algorithm for monitoring parametric deterministic string rewriting systems, to serve as an efficient runtime verification technique for specifying and monitoring arbitrarily complex properties; indeed, string rewriting systems are known to be as expressive as Turing machines [44]. We also provide an implementation of our algorithm as an MOP *logic plugin* [62, 58], so it can be used as an integral part of the JavaMOP runtime verification system. This finally gives JavaMOP the ability to monitor any possible safety property with a formal specification. By abuse of vocabulary, we will refer to deterministic string rewriting systems as string rewriting systems and abbreviate them SRSs.

13.1.1 Examples

The JavaMOP specification presented in Figure 13.1, which uses the new **srs** logic plugin, is able to catch situations in which a monitored program attempts to write to a closed `FileWriter`. JavaMOP specifications begin with a declaration of the name of the specification and parameters. Here the property is named `SAFEFILEWRITER`, and one parameter `f` of type `FileWriter`. The parameters allow us to associate separate monitor states with each object instantiation of the parameters. In this case, with one parameter, there will be one monitor state associated with each object instance of `FileWriter`. This is important because we would not want calls to different object instances of the `FileWriter` class to interfere with each other.

The next part of a JavaMOP specification is the declaration of events. Here we are able to generate three different events: `open`, `write`, and `close`. The events are defined using a superset [199] of AspectJ [164] advice with embedded pointcuts. Here, the event `open` occurs when the `FileWriter` constructor is called.

After the event definitions, we list the formalized property. The keyword **srs** tells JavaMOP that the following property will be a deterministic string rewriting system. Rules in our SRS formalism take the form “ $l \rightarrow r$.”, meaning that the string of events on the left hand side of the arrow rewrites to that on the right side. We only need two rules to specify our desired property. The first rule states that we replace `open write` with `open`. This allows us to collapse multiple safe write operations. The second rule

```

SafeFileWriter(FileWriter f) {

    event open after() returning(FileWriter f) :
        call(FileWriter.new(..)) {}
    event write before(FileWriter f) :
        call(* write(..) && target(f) {})
    event close after(FileWriter f) :
        call(* close(..) && target(f) {})

    srs :
        open write -> open .
        close write -> #fail .

    @fail {
        System.out.println("write after close");
    }
}

```

Figure 13.1: SAFEFILEWRITER SRS SPECIFICATION

catches our misuse case: when we have a write after a close. Here we use one of the three special keywords in the SRS plugin, `#fail`, that signifies that a failure has occurred. Also available is `#succeed`, which allows for denoting success, and `#epsilon`, which simply deletes the left-hand side of a rule from the current string of events.

Note that the SRS rules can be applied in any order when a new event is received, so it is user's responsibility to write *confluent* SRSs or to use the deterministic order of rule application explained in Section 13.3.1. The last part of a JavaMOP specification is the handler section. Handlers are arbitrary Java code that is executed when the monitor raises a particular condition. Here the keyword `@fail` denote that the code within the subsequent braces is run when the string rewrites to `#fail`. Using `@succeed` works respectively for the `#succeed` keyword. The SRS algorithm allows any arbitrary handler keyword other than `#epsilon`. In this example, the handler simply prints out an informative message when an invalid write occurs. In general, handler code may be used for anything, such as running a specific algorithm or recovering from the error denoted by the failure of the safety property in question.

Here we show two further examples of safety policies expressed using

deterministic SRS. We show only the property without worrying about the definition of events or handlers for the sake of brevity. The first property, called HASNEXT, is a property of the Java `Iterator` interface stating that `hasNext()` should always be called and return `true` before `next` is called. Below it is specified as a regular expression:

$$(\text{hasnexttrue next})^* \text{next}$$

The corresponding SRS is as follows:

$$\begin{aligned} \text{hasnexttrue next} &\rightarrow \# \text{epsilon} \\ \text{hasnexttrue hasnexttrue} &\rightarrow \text{hasnexttrue} \\ \text{^next} &\rightarrow \# \text{fail} \end{aligned}$$

While this SRS is certainly larger than the original ERE, it may be easier to understand by some users because it directly captures the semantics of the property by simply enumerating all the cases that one has to worry about. The rule `hasnexttrue hasnexttrue → hasnexttrue` conveys the notion that multiple calls to the `hasNext()` method are idempotent. `hasnexttrue next` rewrites to `#epsilon` because it is a safe operation. If `next` is seen at the beginning of the string a failure is raised as `hasnexttrue` was not properly called. Because our algorithm is incremental and deterministically rewrites from left to right it is not strictly necessary to match the beginning of the string, but it is more clear conceptually.

The second property is called SAFELock which corresponds to the proper nesting of acquiring and releasing locks. Proper nesting, in this case, means that corresponding calls to `acquire()` and `release()` occur within the same method body. Here `begin` and `end` denote the beginning and end of a method body.

$$\begin{aligned} S &\rightarrow \epsilon \mid S \text{ acquire } M \text{ release } A \\ M &\rightarrow \epsilon \mid M \text{ begin } M \text{ end } \mid M \text{ acquire } M \text{ release} \\ A &\rightarrow \epsilon \mid A \text{ begin } \mid A \text{ end} \end{aligned}$$

The property is fairly complex, and a complete explanation can be found in [196]. The SRS for the property follows:

$$\begin{aligned} \text{begin end} &\rightarrow \# \text{epsilon} \\ \text{acquire release} &\rightarrow \# \text{epsilon} \\ \text{begin release} &\rightarrow \# \text{tooManyReleases} \\ \text{acquire end} &\rightarrow \# \text{tooFewReleases} \end{aligned}$$

In this case, the SRS is quite a bit less complex than the context-free grammar specifying the same safety property. Again, it conveys interesting semantic information. From the SRS it is clear that a `begin` followed immediately by a `release()` results in an error because we require all `release()` to occur in the same method call as the corresponding `acquire()`. Similarly, an `acquire()` follow by a `end` results in an error because the lock is not correctly released within the method body. `begin end` and `acquire release` rewrite to `#epsilon` because they are properly nested when they occur adjacently. The SRS is also able to encode information that cannot be encoded in the context-free grammar. By using the handlers `#tooManyReleases` and `#tooFewReleases` we are able to run different handlers when there are too many or too few releases, respectively. This allows us to, for example, ignore an extraneous release or to add a missed release into the control stream. This cannot be done with the context-free grammar without adding extra Java code to the MOP specification.

13.1.2 Contributions

There are two main contributions to this paper:

- An efficient, optimized string rewriting algorithm. It builds upon a modification of the Aho-Corasick algorithm [6]. The original algorithm was designed for quickly finding strings in text. Our modified algorithm keeps track of substitution boundaries so that a rewrite step can be performed in time linear to the length of the right hand side of the matched rule². To our knowledge, this is the first time it has been applied to string rewriting. An optimization has also been devised, which checks for early termination of rewriting.
- An implementation and extensive evaluation of the above algorithm as an MOP logic plugin for runtime verification. This way, it can serve as a specification formalism for parametric safety properties in instances of the MOP framework, such as JavaMOP. We show that its performance in practical runtime verification of large systems is acceptable when compared to other means to specify the same properties. Additionally, we show that it outperforms one of the state-of-the-art rewrite engines, Maude [71], which implicitly supports string rewriting as rewriting modulo associativity.

²The right hand side must be copied, so that the rule is still viable the next time it matches.

13.1.3 Paper Outline

Section 13.2 presents related work in the field of runtime verification: popular runtime monitoring systems, with a particular emphasis on those with greater than finite state specification languages and on our framework of choice for our implementation and experimental test-bed, MOP. Section 13.3 presents our string rewriting algorithm, with its use and construction of pattern match automata and optimization that allows for early termination. Section 13.4 presents our experimental results, and Section 13.5 concludes.

13.2 Related Work and MOP

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to [199] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired, and few of them share the same logic.

There are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces). The handlers specify what actions to perform under exceptional conditions; such conditions include violation and/or validation of the property. It is worth noting that for some logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula. MOP allows for handlers for any number of user defined exceptional situations (called handler categories).

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Figure 13.2, which shows an (incomplete) summary of runtime monitoring systems. For example, JPax can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP (the Java instance of MOP) has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Approach	Logic	Scope	Mode	Handler
JPaX [126]	LTL	class	offline	violation
TemporalRover [88]	MiTL	class	inline	violation
Monopoly [33]	MFOTL	global	offline	validation
Larva [77]	DATE (timed automata)	multiple	online	transitions
JavaMaC [166]	PastLTL	class	outline	violation
Hawk [83]	Eagle	global	inline	violation
RuleR [31]	RuleR	global	inline	violation
Tracematches [25]	Reg. Exp.	global	inline	validation
J-Lo [38]	LTL	global	inline	violation
Pal [52]	modified Blast	global	inline	validation
PQL [192]	PQL	global	inline	validation
PTQL [118]	SQL	global	outline	validation

Figure 13.2: A Selection of Monitoring Systems

Of the systems mentioned in Figure 13.2, only PQL [192], Hawk/Eagle [83], and RuleR [31] provide logical formalisms with greater than finite-state power. Hawk/Eagle adopts a Turing-complete fix-point logic, but it has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available³. Because of this and the fact that Hawk/Eagle has not been run on DaCapo [36] with the same properties, we cannot compare JavaMOP with our new string rewriting systems plugin with Hawk/Eagle. RuleR is a rule-based monitoring system which has the ability to also specify Turing complete properties. The current implementation of RuleR is not built for efficiency, and is, additionally, not publicly available. PQL is not Turing-complete, and performance comparisons with PQL using an older, less efficient, version of JavaMOP can be found in [196]. String rewriting was used in the context of monitoring for detection of malware in [35]. This was, in many ways, the inspiration for adding string rewriting to MOP. However, the string rewriting patterns allowed in that work were regular (i.e., can capture only regular languages), while our goal is to provide a true Turing-complete logical

³[25] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [83]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [41]. We have confirmed the inefficiency claims of [25] with the authors of Hawk/Eagle.

formalism for parametric monitoring.

MOP [62, 58] is an extensible runtime verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an instance of MOP for the Java programming language. It allows the developer to specify desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system.

MOP is a highly extensible and configurable runtime verification framework. The user is allowed to extend the MOP framework with his/her own logics via *logic plug-ins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture [58], which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. MOP also provides efficient and logic-independent support for *parametric* parameters [57], which is useful for specifying properties related to groups of objects. This extension allows associating parameters with MOP specifications and generating efficient monitoring code from parametric specifications with monitor synthesis algorithms for non-parametric specifications. MOP's generic support for parametric patterns simplified our SRS plug-in's implementation.

The JavaMOP instance provides two interfaces: a web-based interface and a command-line interface, providing the developer with different means to manage and process JavaMOP specifications. AspectJ [164] is employed for monitor integration: JavaMOP translates outputs of logic plug-ins into AspectJ code, which is then merged within the original program by an AspectJ compiler. Seven logic-plug-ins are currently provided with JavaMOP: finite state machines, extended regular expressions, context-free grammars, past time linear temporal logic, linear temporal logic with past and future operators, past time linear temporal logic with calls and returns, and, now, string rewriting systems. Descriptions of the first six plugin-ins can be found in [199].

13.3 Monitoring SRS Specifications

In this section, we present some basic notation for string rewriting systems and our string rewriting algorithm which was implemented as a logic plugin in the MOP framework.

13.3.1 Preliminaries

We refer the reader to [44] for an in-depth presentation of string rewrite systems. For an alphabet Σ , a *string rewriting system* (SRS) is a binary relation, R , on Σ , that is, a subset of $\Sigma^* \times \Sigma^*$. The set $\{l \in \Sigma^* \mid (l, r) \in R\}$ is called the *domain* of R , denoted $dom(R)$, while similarly the set $\{r \in \Sigma^* \mid (l, r) \in R\}$ is called the *range*, denoted $range(R)$. We refer here to any element $(l, r) \in R$ as a *rule* in R , any $l \in dom(R)$ as a *left hand side (LHS)* of a rule in R , and any $r \in range(R)$ as a *right hand side (RHS)* of a rule in R . In our SRS specifications in this paper and in JavaMOP, rules $(l, r) \in R$ are written using the earlier shown syntax “ $l \rightarrow r$ ”.

The *single-step reduction relation* on Σ^* that is induced by R is defined as: for any $u, v \in \Sigma^*$, $u \rightarrow_R v$ if and only if there exists $(l, r) \in R$ such that for some $x, y \in \Sigma^*$, $u = x l y$ and $v = x r y$. The *reduction relation* on Σ^* induced by R is the reflexive, transitive closure of \rightarrow_R and is denoted by \rightarrow_R^* . If for $x, y \in \Sigma^*$, $x \rightarrow_R^* y$ and y is irreducible, y is a *normal form* for x . R is *confluent* if there is only one such y for any given x , regardless of the order in which rules are applied.

In our SRSs in MOP, the symbols $s \in \Sigma$ correspond to either *events* of our property or symbols that appear in the RHS of rules in R . We call our string rewriting systems *deterministic* because the same normal form will always be chosen in the presence of a non-confluent R . Specifically, rules are applied left-to-right, with the smallest rule matching first in the case of overlap (e.g., for LHSs $a a$ and $a a b$, the rule with $a a$ as its LHS will always be applied first, starving the other rule). In the case of a conflict that is not resolved by the above, the order of rules in the SRS specification is used to determine which rule to apply (e.g., if two rules have the same LHS, the one specified first will always be applied).

13.3.2 String Rewriting Algorithm Overview

There are two major parts to our SRS algorithm:

1. Finding matches of the LHSs of rules; and
2. Performing replacements with RHSs of rules.

To make replacements as efficiently as possible, the string of events/symbols that we rewrite is a linked list of the `SpliceList` class, which was specially created for our purposes to allow constant time replacement of a section of the list with another list (splicing). The `SpliceList` class has a special type of

iterator defined for it, called the `SLiterator`, that does *not* follow the normal iterator interface in Java.

Rather than only having `next()` and `hasNext()` methods, the `SLiterator` has `next(int i)`, which moves the `SLiterator` forward `i` times and returns true if it is successful (i.e., does not reach the end of the `SpliceList`), and `get()`, which returns the current element that the `SLiterator` points to. `SLiterator` also has a method, `splice(SLiterator second, SpliceList replacement)`, which takes another `SLiterator` to the same `SpliceList` and replaces the sequence denoted by those two `SLiterators`, inclusively, by a specified sequence replacement. It is because of the inclusive nature of the `splice` method that the `SLiterator` must have a method to retrieve its current element without advancing. The `splice` method makes it imperative for our string matching algorithm to maintain `SLiterators` to the beginning and end of the current LHS under consideration.

In Section 13.3.3 we discuss how this matching occurs using a modification of the Aho-Corasick string searching algorithm [6] that, unlike the base algorithm, keeps track of the beginning of a match, so that rewrites can be performed in constant time (after copying the RHS in time proportional to its length). To make the paper self-contained, we give all the necessary information regarding the Aho-Corasick algorithm, rather than only this modification, but the modification is clearly delineated. To our knowledge, this is the first time any variation on the Aho-Corasick algorithm has been used in string rewriting, and no implementations of SRSs exist, that we could find. In Section 13.3.4, we present an in-depth explanation of how the pattern matching fits into the string rewriting algorithm and how we optimize string rewriting to avoid considering sequences that cannot match any LHS.

13.3.3 Pattern Match Automata

The pattern match automata used by our string rewriting process, as mentioned, is a modification of the Aho-Corasick algorithm for finding strings in text [6]. The Aho-Corasick algorithm, which was originally not designed for string rewriting, is able to find all matches in a string in one linear pass, rather than performing separate passes for each rule LHS as would a naive matching algorithm. Our modification of the algorithm allows us to correctly adjust the `SLiterator` to the beginning of our current match, facilitating quick rewrites.

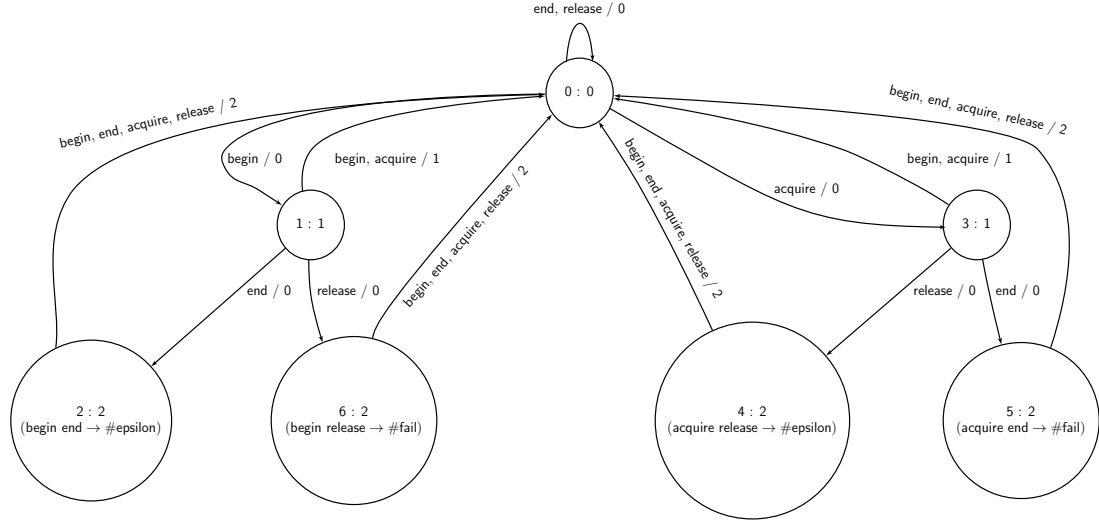


Figure 13.3: Pattern match automaton for the SAFELOCK property (see Section 15.2)

Using Pattern Match Automata

Figure 13.3 shows the pattern match automaton for the SAFELOCK property. Each node has at least its state number and state depth, listed as a pair *number:depth*. The depth is used in two places in the automata generation algorithm, and simply states how many symbols (events) have been processed since the start state in one of the LHSs of the rewrite rules in our SRS. This will be explained in more detail below. Additionally, states which correspond to matching the left hand side of a given rule also display that rule, e.g., in state 6, the `begin release → #fail` rule is matched. Each edge is marked by the list of symbols that cause that transition, as well as a number following a “/”. That number, which we refer to as the *action*, is the number of times to increment the *first* SLiterator except in the self-transitions of state 0. When a self-transition in state 0 occurs, the *first* iterator must be incremented once. When a forward transition with “/ 0” is encountered, a transition to the next state is made, and the next input is considered. If the transition is suffixed with something *other* than 0, the transition must be a backward transition, and the same symbol that is currently under consideration must be evaluated in the next state. This is why we handle self-transitions in state 0 as a special case, if it were suffixed with “/ 1” and handled as a backward

transition, the same symbol would be considered infinitely.

Figure 13.4 shows the pseudocode for pattern matching using a given pattern match automaton. The only global variable for the algorithm is the given `PatternMatchAutomaton`, *pma*. The algorithm begins by initializing the *first* and *second* `SLiterators` to the beginning of the argument `SpliceList l`, using the *head()* method. The local *currentState* is initialized to the initial machine state, here represented as 0⁴. The while loop beginning on line 10 will only exit when the end of *l* is reached, denoted by the `break` statements on lines 20 and 25. We know that the end of *l* is reached on lines 20 and 25 when the `next(int i)` method returns false. We never need to check if *first.next* returns false because it may never advance past *second* due to the construction of the `PatternMatchAutomaton`. Lines 17–22 cover the self transition to state 0 mentioned earlier, while lines 23–27 represent a normal forward transition. 23–27 are a forward transition because the *action* of the transition is 0. As mentioned earlier, the only difference between the 0 self-transition and a forward transition is that in the self-transition the *first* `SLiterator` need be incremented (line 18). Lines 28–30 handle a backward transition in the `PatternMatchAutomaton`. As expected, with a backward transition the *first* `SLiterator` is incremented a number of times specified by the *action* of *transition* and *second* is *not* incremented so that the same symbol will be considered in the next iteration of the loop. One interesting property of this algorithm is that if one pattern is a prefix of another, such as the patterns “*a a* \rightarrow *c*” and “*a a b* \rightarrow *d*”, both matches will be reported. This is undesirable behavior for rewriting because “*aa*” will be rewritten to *c* immediately and “*a a b*” should no longer be matchable. This will be accounted for in Section 13.3.4.

As an example of how the pattern match algorithm functions, suppose that the following series of events have been seen at a given point in a program: `begin begin acquire begin end`. At this point, the `SAFELOCK` property will experience its first match of a rule LHS. Figure 13.5 shows the state transitions as each symbol is considered, as well as the position of the *first* `SLiterator`. An important thing to note is that every time we transition back to state 0, the *first* `SLiterator` index is incremented by 1 (specified by the back transitions), and the symbol is evaluated again in state 0. In general, back transitions need not be to state 0, as we shall see. At the end of the input, the algorithm is in state 2, which matches the rule `begin end \rightarrow #epsilon`. The *first* `SLiterator` correctly points to index 3, which is the last `begin` event. The

⁴It is actually a class that may contain a matched rule, as we can see in Figure 13.3.

```

1  globals PatternMatchAutomaton pma
2  locals SIterator first, second
3      State currentState, nextState
4      Symbol symbol
5      Transition transition
6  procedure match(SpliceList l)
7  first  $\leftarrow l.head()$ 
8  second  $\leftarrow l.head()$ 
9  currentState  $\leftarrow 0$ 
10 while (true){
11   if (currentState.hasMatch()){
12     //signal match
13   }
14   symbol  $\leftarrow second.get()$ 
15   transition  $\leftarrow pma.get(currentState, symbol)$ 
16   nextState  $\leftarrow transition.state$ 
17   if (nextState = 0){
18     first.next(1)
19     if ( $\neg second.next(1)$ ){
20       break
21     }
22   }
23   else if (transition.action = 0){
24     if ( $\neg second.next(1)$ ){
25       break
26     }
27   }
28   else {
29     first.next(transition.action)
30   }
31   currentState  $\leftarrow nextState$ 
32 }

```

Figure 13.4: Pattern Match Algorithm

current state	symbol	next state	<i>first</i> index
0	begin	1	0
1	begin	0	1
0	begin	1	1
1	acquire	0	2
0	acquire	3	2
3	begin	0	3
0	begin	1	3
1	end	2	3

Figure 13.5: A run of the pattern match algorithm on begin begin acquire begin end

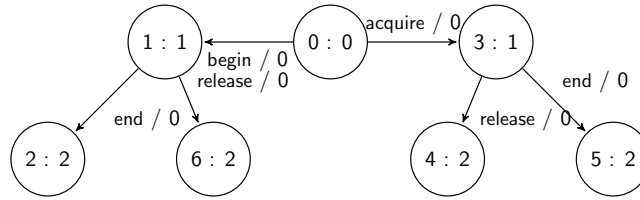


Figure 13.6: Forward Transitions for SAFELock (matched rules omitted)

second SLiterator always points at the current input, which is end. These SLiterators can then be used to quickly replace begin end with #epsilon, as we will see in Section 13.3.4.

Generating Pattern Match Automata

There are two main phases to the creation of pattern match automata. In the first phase the forward transitions of the automaton are created. In the second phase, all of the backward transitions and the self-transition that (almost) always exists in state 0 are added. During the computation of the backward transitions, the actions for the backward transition are also computed and added to the backward transitions. As mentioned, only backward transitions ever have non-0 actions, since they correspond to places in the automaton where there is a switch from matching one potential set of LHSs of rules to another. For instance, in Figure 13.5, between the third

begin and the first acquire, there is a switch from potentially matching $\{\text{begin end, begin release}\}$ to $\{\text{acquire end, acquire release}\}$, which requires no longer considering the begin event for match purposes, thus the action of 1.

To create the forward transitions for an automaton, we add one path that corresponds directly to the left hand side of each rule in our string rewriting system. We add these paths one at a time, and reuse as many states as possible. Each forward transition is assigned the action 0. Figure 13.6 shows the forward transitions for the pattern match automaton originally presented in Figure 13.3. For each LHS, we begin at state 0 and add a transition for the first symbol. Because all patterns SAFELock begin with either `begin` or `acquire`, we have only two transitions, one labeled with `begin` and one labeled with `acquire`. We continue to transitively add transitions based on the remainder of each LHS. For the two rule LHSs beginning with `begin`, one ends with `end` and the other ends with `release`, so there are two transitions out of state 1 labeled accordingly. As each new state is added to the machine during the forward transition phase, the depth of the state is recorded. The depth is simply the number of symbols from state 0. For instance, state 6 is at depth 2, since two symbols, `begin` followed by `end`, lead to state 6. The largest depth always corresponds to the longest rule LHS.

In the second phase, the self-transition on state 0 is added first, if needed. The self-transition is only necessary if there is not a forward transition out of state 0 for every symbol used in the SRS or specified by the JavaMOP front end⁵.

After potentially adding the self-transition in state 0, the backward transitions are added to the pattern match automaton. Backward transitions are only added from a given state for symbols that do not have forward transitions out of that state. All backward transitions from a given state, s , will go to the same place, so we define $fail(s) = s'$, where s' is the destination of a backward transition out of s . To find the destination for the backward transitions out of a state in pattern match automaton pma with depth d , we consider each state r of depth $d - 1$ and perform the following actions, transitions are added in depth first order [6]:

1. If $pma.get(r, a)$ is a backward transition for all symbols a , do nothing.
2. Otherwise, for each symbol a such that $pma.get(r, a) = s$, do the following:

⁵JavaMOP allows one to define events that do not appear in the specified property; these will correspond to symbols that are never rewritten by the specified SRS.

- (a) Let $s' = fail(r)$.
- (b) Compute $s' \leftarrow fail(s')$ until such point as $pma.get(s', a).action = 0$. Because state 0 must have either a forward transition or a self-transition for every symbol, such an s' must exist.
- (c) For all a' such that $pma.get(s, a')$ has no forward transition, assign $pma.get(s, a').state = s'$, **$pma.get(s, a').action = s.depth - s'.depth$** .

The procedure above is essentially the same as [6]. The part in bold is specific to our algorithm for string rewriting. The action is assigned as such because the depth of a given state represents the number of symbols processed since state 0 in the automaton, thus the difference in the depths tell us the number of symbols that we need to skip with the *first* SIterator in Figure 13.4. While the pattern match automaton for SAFELOCK has backward transitions that only go to state 0, as mentioned, this is not always the case in general. When the suffix of one LHS overlaps with the prefix of another, backward transitions that do not go back to state 0 are generated. An example of this can be seen in Figure 13.7, where the SRS in question is $b\ a\ a \rightarrow \#epsilon$, $a\ a\ c \rightarrow \#epsilon$. Because $b\ a\ a$ and $a\ a\ c$ have a suffix/prefix overlap, the backward transitions from state 3 at depth 3 go to state 5 at depth 2, resulting in an action of only 1. For example, consider input $b\ a\ a\ c$. When we switch from matching $b\ a\ a$ to matching $a\ a\ c$, which occurs between states 3 and 5, we wish to only “forget” the b at the beginning, an action of 1. Note that when we use the rule application order of “smallest left hand side”, this transition will never be taken because $b\ a\ a$ will be immediately rewritten. We include these transitions in the automaton for future rewrite orders.

13.3.4 Rewriting using Pattern Match Automata

The rewriting algorithm we use to monitor SRS’s is presented in Figure 13.8. Not pictured in Figure 13.8, is the action of the monitor itself. As any monitoring algorithm in the MOP framework, events arrive one at a time. As each event occurs, we add it—as a symbol representing that event—to a SpliceList that contains the results of rewriting previous sequences of events. Additionally, if any rules make use of the $\hat{\ }$ symbol, it will be added to the beginning of the SpliceList and treated as a normal symbol by the rewriting

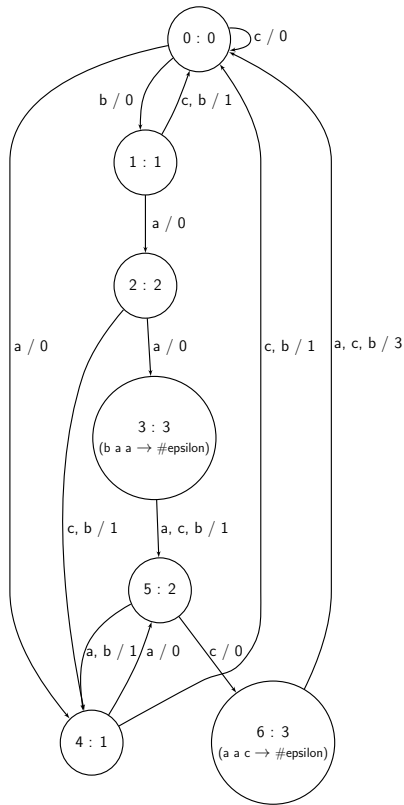


Figure 13.7: A pattern match automaton with overlap

```

1  globals PatternMatchAutomaton pma
2  locals SLLiterator first, second, last
3      State currentState, nextState
4      Symbol symbol
5      Transition transition
6      '='changed pastLast
7  procedure match(SpliceList l)
8  do {
9      first  $\leftarrow$  l.head()
10     second  $\leftarrow$  l.head()
11     currentState  $\leftarrow$  0
12     changed  $\leftarrow$  false
13     pastLast  $\leftarrow$  false
14     while (true){
15         if (currentState.hasMatch()){
16             if (currentState.match = #succeed){
17                 // raise succeed
18             }
19             if (currentState.match = #fail){
20                 // raise fail
21             }
22             first.splice(second, currentState.match)
23             nextState  $\leftarrow$  0
24             changed  $\leftarrow$  true
25             pastLast  $\leftarrow$  false
26             last  $\leftarrow$  second
27             second  $\leftarrow$  first.copy()
28         }
29         symbol  $\leftarrow$  second.get()
30         transition  $\leftarrow$  pma.get(currentState, symbol)
31         nextState  $\leftarrow$  transition.state
32         if (nextState = 0){
33             first.next(1)
34             if ( $\neg$ second.next(1)){
35                 break
36             }
37         }
38         else if (transition.action = 0){
39             if ( $\neg$ second.next(1)){
40                 break
41             }
42         }
43         else {
44             first.next(transition.action)
45         }
46         if ( $\neg$ changed){
47             if (second = last){
48                 pastLast  $\leftarrow$  true
49             }
50             if (pastLast and nextState = 0){
51                 return
52             }
53         }
54         currentState  $\leftarrow$  nextState
55     }
56 } while (changed)

```

Figure 13.8: Rewriting Algorithm

algorithm. As for uses of \$, the current event must be added *before* \$⁶.

After an event is added to the `SpliceList`, the algorithm in Figure 13.8 is evaluated to completion before another event can be accepted. The algorithm is similar to the pattern match procedure of Figure 13.4. The changes are in bold. There are three main changes: the inclusion of a loop that ensures that a normal form is reached, the actual rewriting step itself, and a section that recognizes early termination.

The first new control structure to notice is the `do...while` loop from line 8 to 56. This loop ensures that rewriting continues until there is a pass through the loop in which nothing changes, i.e., the string is in normal form. The new boolean variable, *changed*, controls this loop. It is set to false at the beginning of an iteration of the `do...while` loop, and to true on line 24, which is only executed when a rewrite occurs.

Lines 15–28 perform the actual rewriting step. The element *match* of a `State` contains the right hand side of the rule matched in that `State`. If the *match* is one of the two special keywords `#succeed` or `#fail`, a success or fail handler is executed, as appropriate, and rewriting terminates. If either handler is executed, the monitor is considered dead unless it is reset (see [199]). If *match* is something else, the `splice` method is called on line 22. The `splice` method is a special method of `SLiterator` that replaces a range specified by the *this* and an argument `SLiterator` with the argument sequence. Here the range is specified by *first* and *second*, and *currentState.match* is passed as the replacement. Note that if the right hand side of the rule is `#epsilon`, it is represented as an empty sequence, which `splice` is able to handle. The `splice` method also correctly sets *first* and *second* to point to the beginning and end of the spliced in *match* sequence, or the next symbol if *match* was `#epsilon`. On line 26, we set *last* to *second*, so that *last* points to the end of the last replacement, this will be used to determine early termination. Then, on line 27, *second* is set as a copy of *first*. This ensures that segments of string which are transitively rewritten will be rewritten immediately. Because `splice` changes the `SpliceList`, it is important to set *currentState* back to state 0 because any matching will occur in the newly rewritten segment of the `SpliceList`.

In the last new addition to the match algorithm, from lines 46 to 53, we test for early termination of the algorithm. The idea here is to exit early if we enter a segment of the `SpliceList` that we know for certain cannot be

⁶Because of this there is a very small performance hit for using \$ in a rule, but ^ is essentially free.

event	initial l	l in normal form
begin	begin	begin
end	begin end	#epsilon
begin	begin	begin
acquire	begin acquire	begin acquire
release	begin acquire release	begin
acquire	begin acquire	begin acquire
end	begin acquire end	#fail

Figure 13.9: An SRS monitoring run for SAFELock

rewritten. This happens when we reach a point that is past the end of the *last* SIterator, which was set in a previous iteration, no rewrites have occurred in the current iteration, and *currentState* returns to 0. The first two requirements are fairly straight-forward: if a change occurs, new matches are possible, and if we are in a segment of the SpliceList before the last rewrite, we are still investigating symbols that are potentially new. However, if there is no rewrite in the current iteration and we are past the last change from the previous iteration, we are seeing symbols that were seen in the previous iteration with no change. The last condition, that we must return to State 0, is more subtle. The reason for this is that there could have been a rewrite in the last iteration that inserted a segment that appears in the middle of a left hand side of one of the rules. A simpler way to look at this requirement is that if *pma* is not in state 0 it is actively matching *something*. This condition for early termination can lead to an unbounded amount of saving, as the SpliceList can be of an unbounded length.

Figure 13.9 shows a monitor run as non-parametric events for SAFELock arrive. The non-parametric events are dispatched to the correct monitor instance by the indexing of JavaMOP (or whatever projection method is used in future language instances of MOP). The first column shows the arriving event, the second column shows the state of the SpliceList l before any rewriting, and the last column shows the normal form for l after the rewriting algorithm of Figure 13.8 has run. After the last event a failure has occurred, and the fail handler will execute.

Benchmark	Orig. (ms)	HASNEXT		SAFE SYNC COL		SAFE SYNC MAP		UNSAFE ITER		UNSAFE MAP ITER	
		ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS	ERE	SRS
avroa	2317*	194	227	35*	103*	28	120*	253*	288*	41	134*
batik	773	0	6	5	11	9	-1	5	3	1	2
eclipse	11749	-1	-2	-2	-4	-1	-3	-2	-2	-2	-2
fop	251	922	2091	26	24	21	20	34	57	28	42
h2	3860	9	15	6	2	0	4	15	22	8	24
jython	1400	3	4	3	3	4	2	16	18	3	3
luindex	478	2	-1	2	0	0	4	1	2	0	-5
lusearch	581	1	3	-1	1	3	3	46	46	2	0
pmd	1441	27	117	139	137	10	17	72	148	177	199
sunflow	1222	5	8	0	-1	6	-3	-4	4	0	3
tomcat	1068	2	4	3	3	3	1	2	2	2	2
tradebeans	4618	2	1	-1	-3	-1	2	4	-2	-2	-1
tradesoap	3213	1	-1	1	-1	0	-2	1	0	0	0
xalan	359	5	1	5	1	6	3	90	172	7	8

Figure 13.10: Comparison of JavaMOP with extended regular expressions (ERE) and with the same properties expressed as string rewriting systems (SRS): average *percent* overhead (convergence within 3% except those marked with *)

13.4 Evaluation

An important thing to note about SRS execution is that it may add an unbounded amount overhead to a program execution in full generality, since it is Turing-complete. Because of this, our evaluation focuses on two specific types of experiments: first we show how it compares, within the context of JavaMOP, to finite-state logics on the DaCapo benchmark suite [36]. Then we give a comparison of our underlying SRS rewrite engine against the Maude [71] term rewriting engine, modulo associativity. The goal of the first evaluation is to show that SRS monitoring is efficient enough to be used in large programs, being not much less efficient than finite-state logics⁷ when monitoring finite-state properties. However, it should be stressed that a fully recursively enumerable safety property may add an unbounded amount of overhead, or even not terminate. The goal of the second experiment is to show that our SRS implementation is more efficient than the state-of-the-art⁸.

All experiments were performed on a machine with a 3.82GHz Intel® Core™ i7 970 hexcore with Hyper-Threading (12 hardware threads) and 24 GB of ram. Ubuntu 11.10 64 bit was used as the operating system and version 9.12 of DaCapo was used as the benchmark suite, with default inputs and the -converge option to gain convergence within 3%. OpenJDK version 1.6.0.23 as the Java virtual machine. All compiled JavaMOP specs were weaved into DaCapo using ajc 1.6.11. Maude 2.6 was used for comparison with Maude.

The following properties were used in the DaCapo experiments. The SRS versions of them (shown below) are new, while the extended regular expression versions were borrowed from [41, 40, 57, 196].

- HASNEXT: Do not use the next element in an Iterator without checking for the existence of it (see Section 13.1.1);
- SAFESYNCCOL: If a Collection is synchronized, then its iterator also should be accessed synchronously:

<code>sync asyncCreateliter</code>	<code>→</code>	<code>#fail</code>
<code>sync syncCreateliter accesslter</code>	<code>→</code>	<code>#fail</code>

⁷In this case, extended regular expressions.

⁸Note that Maude is more general than our SRS engine, but there is a price for that generality, and general term rewriting makes little sense in the context of MOP event traces.

N	Maude Time (ms)	SRS Time (ms)
100	42	33
1000	37038	236
5000	DNF	7112
10000	DNF	26132

Figure 13.11: Comparison of maude versus SRS rewrite. DNF: did not finish in one hour

- **SAFESYNCMAP**: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner:

`sync createSet asyncCreateltr` \rightarrow `#fail`
`sync createSet syncCreateltr accessltr` \rightarrow `#fail`

- **UNSAFEITER**: Do not update a Collection when using the Iterator interface to iterate its elements:

`update use` \rightarrow `#fail`
`use use` \rightarrow `use`
`update update` \rightarrow `update`
`createltr` \rightarrow `#epsilon`

- **UNSAFEMAPITER**: Do not update a Map when using the Iterator interface to iterate its values or its keys:

`update use` \rightarrow `#fail`
`use use` \rightarrow `use`
`update update` \rightarrow `update`
`createltr` \rightarrow `#epsilon`
`createCollection` \rightarrow `#epsilon`

For the comparison with Maude, strings of equal numbers of 2's, 1's, and 0's, with the 2's preceding the 1's preceding the 0's were generated, and the following rewrite system applied. Note, that the language of strings that reduce to `#epsilon` with this rewrite system is strictly context-sensitive.

`1 0` \rightarrow `0 1` `2 0` \rightarrow `0 2`
`2 1` \rightarrow `1 2` `0 1` \rightarrow `3`
`1 3` \rightarrow `3 1` `3 0` \rightarrow `0 3`
`3 2` \rightarrow `#epsilon` `2 3` \rightarrow `#epsilon`

Figure 13.10 shows a comparison of finite-state properties specified in JavaMOP using ERE and SRS. The first column shows the individual DaCapo [36] benchmarks, and the second column shows runtime of the original uninstrumented benchmarks in milliseconds. All other columns are *percent* overhead. Each benchmark-property pair converged to within 3% except the instances of *avroa* marked with *. The results presented for *avroa* that did not converge are the average of twenty runs with outliers removed, but they are still not as trustworthy as the converging results. This lack of convergence is a problem on highly multi-threaded machines. We can see that even the uninstrumented, original run, fails to converge. Negative overheads are the result of noise in the experimental settings and changes in code layout due to instrumentation resulting in slightly more efficient programs.

Overall, the average overhead on the DaCapo benchmark suite was 58% for SRS, while it was 33% for ERE. When *fop*-HASNEXT—which has, by far, the worst overhead of any trial—is removed from both, the overhead drops to 29% and 20%, respectively. It must be noted, that the properties we use are specifically selected for generating large overheads; they are very intensive properties that generate *many events* (see [160]). The overhead numbers are slightly larger than reported in previous papers because we have moved to a multi-threaded, and quite simply faster, machine. The monitors in JavaMOP must be synchronized, which results in higher overhead for programs that actually make use of multiple threads. Any monitoring system must do the same thing if the monitors are for cross-thread properties (like all of those properties used here). In most of the benchmark/property pairs, the performance of ERE and SRS are very comparable. For *pmd*-HASNEXT and *avroa*-SAFESYNCMAP, SRS shows more than three times the overhead of ERE, but for all other trials SRS is never more than three times worse.

Figure 13.11 shows the comparison of Maude to our SRS engine with the rewrite system discussed above. N refers to the number of each digit, i.e., N=100 has 300 characters in it: 100 each of 2, 1, and 0. As we can see from the results, our SRS engine runs in 78% of the time of maude at N=100. At N=1000, our SRS engine runs in .006% of the time of Maude. With larger inputs, Maude fails to complete in an hour, while our SRS engine takes less than 30 seconds on every tested input.

Acknowledgments

The work presented in this paper was supported in part by the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

13.5 Conclusion

We provided the first means to efficiently monitor parametric Turing-complete specifications using string rewriting systems. By using a modified version of the Aho-Corasick string matching algorithm and a means to terminate the rewriting process early, the resultant string rewriting algorithm is quite practical, as shown in our extensive evaluation⁹.

The average overhead on the DaCapo benchmark suite was 58% for finite state properties monitored using SRS, while it was 33% using ERE plugin to monitor the same properties. When the largest benchmark/property pair is removed from both, the overhead drops to 29% and 20%, respectively. This shows that the SRS plugin can be efficient when monitoring reasonable properties on large programs. We must stress however that arbitrarily complex properties may add arbitrary overhead, regardless of the efficiency of the monitoring algorithm or implementation there of.

A less extensive comparison of our core string rewriting algorithm with the term rewrite engine Maude, which provides implicit support for string rewriting through its rewriting modulo associativity, suggests that our approach vastly outperforms the state-of-the-art in rewriting, when restricted to string rewriting¹⁰.

Exercises

Exercise 25 *Revise the pattern-match automata and rewriting algorithms in Figures 13.4 and 13.8, respectively. In class, it was suggested that there was a problem in these algorithms with how pattern-matching was done. Can you find a problem, or you think they are correct? Explain with enough detail to show that you understand the algorithm.*

⁹Special thanks to Dongyun Jin for help with DaCapo experimental settings.

¹⁰The full generality of term rewriting provided by Maude makes little sense when applied to monitoring safety properties, which necessarily operate on strings (traces) of events.

Exercise 26 *Manually execute the SRS monitoring algorithm in Figure 13.8 for the SafeLock property specified as an SRS in Section 13.1.1, on the same trace as in Exercise 22: begin begin acquire acquire release release end begin acquire release end end begin end. Explain only the major steps, demonstrating that you understand the SRS monitoring algorithm. Compare, on this execution trace, the SRS monitor with the CFG monitor in Exercise 22. Which one do you find more intuitive? Which one is more efficient?*

Exercise 27 *String rewriting is more general than CFG monitoring. Specifically, we can associate an equivalent string rewrite system to any CFG, and then use the former to do monitoring. Describe the general procedure to translate a CFG to a string rewrite system. Then apply the general procedure manually to the SafeLock CFG property in Exercise 22, and redo Exercise 26 with the resulting SRS. How does the resulting SRS monitor compare with the one in Exercise 26? How about the one in Exercise 22? Comment on the asymptotic complexity of monitoring CFG (Chapter 12) vs monitoring the corresponding SRS.*

Chapter 14

Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis

Abstract: The relationship between two well established formalisms for temporal reasoning is first investigated, namely between Allen's interval algebra (or Allen's temporal logic, abbreviated ATL) and linear temporal logic (LTL). A discrete variant of ATL is defined, called Allen linear temporal logic (ALTL), whose models are ω -sequences of timepoints. It is shown that any ALTL formula can be linearly translated into an equivalent LTL formula, thus enabling the use of LTL techniques on ALTL requirements. This translation also implies the NP-completeness of ATL satisfiability. Then the problem of monitoring ALTL requirements is investigated, showing that it reduces to checking satisfiability; the similar problem for unrestricted LTL is known to require exponential space. An effective monitoring algorithm for ALTL is given, which has been implemented and experimented with in the context of planning applications.

14.1 Introduction

Allen's interval algebra, also called Allen's temporal logic (ATL) in this paper, is one of the best established formalisms for temporal reasoning [170]. It is frequently used in AI, especially in planning. Linear temporal logic (*LTL*) [?] is successfully applied in program verification, temporal databases, and related domains. Despite the widespread use of both ATL and *LTL*, there is no formal and systematic investigation of their relationship. This paper makes a step in this direction. To have a semantic basis for such a relationship, we define a discrete variant of ATL, called Allen linear temporal logic (ALTL), whose syntax and complexity of satisfiability are the same as for ATL, but whose models resemble those of *LTL*.

We show that ALTL can be linearly encoded into a subset of *LTL*. This encoding yields the NP-completeness of the satisfiability problem for an ATL (proposed in [110]) slightly richer than the original one proposed by Allen. On the practical side, this result allows us to use the plethora of techniques and analysis tools developed for *LTL* on requirements (or compatibilities) expressed using ATL. Since ATL is *the* logic of planning, and since validation and verification (V&V) of complex plans for systems with decisional autonomy is highly desirable, if not crucial, in many applications, this automated translation into *LTL* potentially enables us to use well-understood V&V techniques and tools in a domain lacking (but in need of) them. Further, it may also support the suggestion made in [48] that *LTL* can be itself seriously regarded as a suitable formalism for temporal reasoning in AI, and particularly in planning. There are, however, complexity aspects that cannot be ignored (some of them pointed in this paper).

The importance of monitoring in planing cannot be overestimated. For example, an autonomous rover whose execution plans have been rigorously verified may still fail for reasons such as hardware or operating system failures, unexpected terrain in an unknown environment, etc. Having monitors to check online the execution of plans step by step and to trigger recovery code in case of violations is of crucial importance. It is the challenge of generating efficient monitors from planning requirements that motivated the work in this paper. We argue that a blind use of monitoring algorithms for *LTL* to monitor ALTL formulae is not feasible even on small ALTL formulae; then we give a special-purpose monitoring algorithm for ALTL which only needs to call a boolean satisfiability checker at each step if synchronous monitoring is desired, or at the end of the monitoring session if asynchronous monitoring is acceptable, or anywhere in between, for example at specific relevant events,

such as synchronization points. Since checking satisfiability of a formula is a simpler problem than synchronous monitoring (a synchronous monitor should report violation right away if the formula is not satisfiable), the algorithm proposed in this paper is asymptotically optimal. This result is particularly interesting because, for unrestricted *LTL*, it is known that any monitor (synchronous or not) needs exponential space [217]. The proposed monitoring algorithm has been implemented and experimented with in the context of planning for autonomous rovers.

Preliminaries. We assume the reader familiar with Linear Temporal Logic (*LTL*) [?]. We here only recall some basics and introduce our notation. *LTL* is interpreted in “flows of time”, modeled as strict linear orders $(T, <)$, where T is a nonempty set of “time points”. The *LTL* language consists of propositional symbols (p_0, p_1, \dots) , boolean operators (\neg and \wedge), and temporal operators \mathcal{U} (“until”) and \circ (“next”), and *LTL* formulae follow the common syntax $\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathcal{U} \varphi_2 \mid \circ\varphi$. *LTL* models are triples $M = (T, <, v)$ such that $(T, <)$ is a strict total order (a flow of time) and v is a map called valuation associating with each variable p a set $v(p) \subseteq T$ of time points (where p is supposed to be true). The satisfaction relation $M \models \varphi$ is defined as in [?]. Other important temporal operators, such as \Diamond (eventually) and \Box (always), are expressible using \mathcal{U} as $\Diamond\varphi = \text{true} \mathcal{U} \varphi$ (φ will eventually hold) and $\Box\varphi = \neg\Diamond\neg\varphi$ (φ will always hold). \Diamond can also be expressed in terms of \Box , namely $\Diamond\varphi = \neg\Box\neg\varphi$. In this paper we only need the $\{\Box, \Diamond\}$ -fragment of *LTL* (without \circ and \mathcal{U}). Since \Diamond and \Box can be defined in terms of each other, we take the liberty to call this fragment *LTL* $_{\Box}$ (could have also called it *LTL* $_{\Diamond}$). The “satisfiability problem” for a formula φ is concerned with whether there is some model M such that $M \models \varphi$. The satisfiability problem of *LTL* formulae is PSPACE-complete, while the satisfiability of *LTL* $_{\Box}$ is NP-complete [248].

14.2 Allen (Linear) Temporal Logic - ATL (ALTL)

Allen Temporal Logic (ATL) [10] is specified as a framework to deal with incomplete relative temporal information, such as “event A is before or overlaps event B”. Allen takes the *interval* as the primitive temporal quantity and introduces 13 (mutually exclusive) basic binary relations between any two intervals, with the following intuitive meaning: *Equals*(i, j) holds iff i and j consist of the same time points; *Meets*(i, j) (or *MetBy*(j, i)) holds iff j starts *immediately* after i ; *Before*(i, j) (or *After*(j, i)) holds iff i starts

and ends before j , but there is also some proper time elapsed between the end of i and the beginning of j ; $Overlaps(i, j)$ (or $OverlappedBy(j, i)$) holds iff i starts strictly before j starts, they have some common time points, and i ends strictly before j ends; $Contains(i, j)$ (or $During(j, i)$) holds iff j starts strictly after i starts and terminates strictly before i terminates; $Starts(i, j)$ (or $StartedBy(j, i)$) holds iff i and j start together but j continues (strictly) after i ends; dually, $Ends(i, j)$ (or $EndedBy(j, i)$) holds iff i and j terminate together but j starts strictly before i starts. Constraints among intervals, also called requirements or *compatibilities*, are given as boolean combinations of such relations on intervals. In (model-)theoretical works on ATL, time is assumed to flow continuously, typically *not* at an enumerable rate (e.g., timepoints can be rational or real numbers). Following this model, we formally define the semantics of these interval relations in Definition 59; then we propose a time-discrete variant of ATL, in which the time-points are enumerable.

ATL is extensively used in AI planning to formalize and reason about concurrency and temporal extent. In AI planning, intervals can represent both action instances and the states of various attributes or components of a system. Attributes whose states change over time are called *state variables*, each being possibly regarded as a concurrent thread. The history of states of a state variable over a period of time is called a *timeline* and is typically partitioned into *intervals*, where an interval is a set of contiguous timepoints in which the corresponding state variable satisfies some property of interest. A *compatibility* then determines necessary correlations among various behaviors of parts of the system in order for a plan to be legal. One appealing aspect of ATL in this domain is that compatibilities can be elegantly depicted using an intuitive graphical notation (see Figure 14.1), that allows planning specialists to develop surprisingly large and complex specifications in a short time.

Example 17 *We use McCarthy’s classic monkey/banana planning problem as a running example. A monkey is at location “x”, the banana is hanging from the tree. The monkey is at height “Low”, but if it climbs the tree then it will be at height “High”, same as the banana. Available actions are: “Going” from a place to another, “Climbing” (up) and “Climbing Down”, and “Grabbing” banana.*

Attributes. *BANANA has one state variable “Banana-sv” saying if the monkey has the banana or not. LOCATION has one variable “Location-sv” for the location of the monkey. ALTITUDE has one variable “Altitude-sv”*

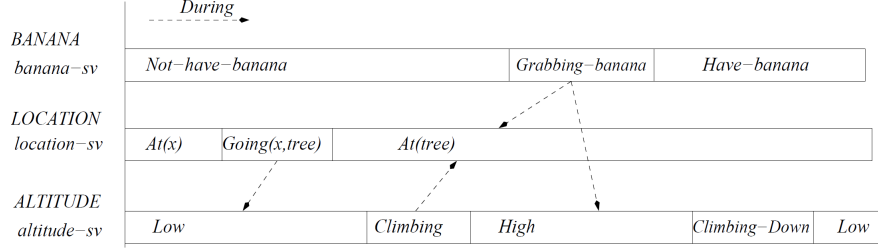


Figure 14.1: Attributes and compatibilities

for the height.

Compatibilities. Now we can consider some compatibilities for the intervals corresponding to these attributes, also depicted in Figure 14.1:

- Have-banana (“ H_b ”) requires Grabbing-banana (“ G_b ”) which requires Not-have-banana (“ N_{hb} ”). Grabbing-banana is performed while High and At(tree).
- At(tree) (“ $@(tree)$ ”) requires going from the location “x” to the tree which requires At(x) (“ $@(x)$ ”). Going(x,tree) (“ $G(x, tree)$ ”) is performed while Low.
- High (“ H ”) requires Climbing (“ C ”) which requires Low (“ L ”), and Climbing-Down (“ C_D ”) requires High. Climbing is performed while At(tree).

These compatibilities can be formally specified in ATL as follows:

$$\begin{aligned}
& \text{Meets}(N_{hb}, G_b) \wedge \text{Meets}(G_b, H_b) \wedge \text{During}(G_b, @(tree)) \wedge \text{During}(G_b, H) \wedge \\
& \text{Meets}(@(x), G(x, tree)) \wedge \text{Meets}(G(x, tree), @(tree)) \wedge \text{During}(G(x, tree), L) \wedge \\
& \text{Meets}(L, C) \wedge \text{Meets}(C, H) \wedge \text{Meets}(H, C_D) \wedge \text{Meets}(C_D, L) \wedge \text{During}(C, @(tree)).
\end{aligned}$$

Let us consider the subformula consisting of the first four conjuncts above (first line), and suppose that an unexpected “flying monkey” wants the banana. It climbs the tree, but it cannot reach for the banana. Being a flying monkey, it jumps for the banana, grabs it while gliding when it is still High and At(tree), but as it glides it leaves the tree location. Supposing that it leaves the tree location at the same time it changes the status from Grabbing-banana to Have-banana, one can notice that the third conjunct is violated. Indeed,

G_b must hold during $@(tree)$, meaning that there must be some (non-zero) periods of time in which the monkey was at the tree location before and after grabbing the banana.

It is often useful to state that some propositions hold all the time or eventually during an interval. For example, assume one more state predicate, *hungry*, saying whether the monkey is hungry or not, and assume that we want to state that monkeys should grab and have bananas only if they are hungry and do not already have bananas. This can be done with the following additional conjunct:

$$Occurs(hungry, N_{hb}) \wedge Holds(hungry, G_b) \wedge Holds(hungry, H_b) \quad \square$$

There are different views on how intervals should be modeled in different time flows. A common interpretation is that the intervals are ordered pairs of distinct points in \mathbb{Q} or \mathbb{R} . For simplicity, it is convenient to use semantics where intervals are arbitrary convex non-empty subsets of time points of an arbitrary time flow.

Definition 56 If \mathcal{P} is a set of **atomic propositions** and \mathcal{I} is a set of **intervals**, then an **Allen temporal logic formula over \mathcal{P} and \mathcal{I}** , or an **ATL(\mathcal{P}, \mathcal{I})-formula** or even just a **formula** when \mathcal{P} and \mathcal{I} are understood from context, is any boolean combination of **basic formulae** of the form $Equals(i, j)$, $Before(i, j)$, $After(i, j)$, $Overlaps(i, j)$, $OverlappedBy(i, j)$, $Meets(i, j)$, $MetBy(i, j)$, $Contains(i, j)$, $During(i, j)$, $Starts(i, j)$, $StartedBy(i, j)$, $Ends(i, j)$, $EndedBy(i, j)$, $Holds(p, i)$, $Occurs(p, i)$, where $i, j \in \mathcal{I}$ and $p \in Bool(\mathcal{P})$.

$Bool(\mathcal{P})$ is the set of boolean propositions over variables in \mathcal{P} . Interestingly, the original formulation of ATL [10] did not include *Holds* and *Occurs*; motivated by practical reasons, they were added later in [110]. To define a formal semantics of ATL we need to first define an appropriate notion of model.

Definition 57 Let $(T, <)$ be a strict total order. The relation $<$ is tacitly extended to a strict partial order on subsets of T , namely $X < Y$ iff $x < y$ for all $x \in X$ and $y \in Y$. Also, by abuse of notation, we may write just x instead $\{x\}$; thus, $x < Y$ means that $x < y$ for all $y \in Y$. For $x, y \in T$ let (x, y) be the set $\{z \in T \mid x < z < y\}$. A subset C of T is **<-convex**, or simply **convex**, iff $(x, y) \subseteq C$ for any $x, y \in C$.

In \mathbb{R} , for example, the convex sets are precisely the intervals. Recall that intervals in \mathbb{R} can be open or closed on any of their ends, and that they may be bound by $-\infty$ or $+\infty$ at their left or right ends, respectively.

Definition 58 A $(\mathcal{P}, \mathcal{I})$ -*interval model* (or simply an *interval model* when \mathcal{P} and \mathcal{I} are understood) is a structure $\mathcal{M} = (T, <, v, \sigma)$, where $(T, <)$ is a strict total order (modeling the intended flow of time), $v : \mathcal{P} \rightarrow 2^T$ is a valuation map assigning to each atomic proposition $p \in \mathcal{P}$ a set of time points $v(p)$ (in which the proposition is assumed to be true), and σ is a map that associates with every interval $i \in \mathcal{I}$ a non-empty convex subset $\sigma(i)$ of T . We may also refer to $(\mathcal{P}, \mathcal{I})$ -interval models as models of $\text{ATL}(\mathcal{P}, \mathcal{I})$.

We are now ready to give the formal semantics of ATL.

Definition 59 An interval model $\mathcal{M} = (T, <, v, \sigma)$ *satisfies*: $\text{Equals}(i, j)$ iff $\sigma(i) = \sigma(j)$; $\text{Before}(i, j)$ or $\text{After}(j, i)$ iff there is some $t \in T$ such that $\sigma(i) < t < \sigma(j)$; $\text{Overlaps}(i, j)$ or $\text{OverlappedBy}(j, i)$ iff $\sigma(i) \cap \sigma(j) \neq \emptyset$ and there are some $t_i \in \sigma(i)$ and $t_j \in \sigma(j)$ such that $t_i < \sigma(j)$ and $\sigma(i) < t_j$; $\text{Meets}(i, j)$ or $\text{MetBy}(j, i)$ iff $\sigma(i) < \sigma(j)$ and there is no $t \in T$ such that $\sigma(i) < t < \sigma(j)$; $\text{Contains}(i, j)$ or $\text{During}(j, i)$ iff there are some $t_i, t'_i \in \sigma(i)$ such that $t_i < \sigma(j) < t'_i$; $\text{Starts}(i, j)$ or $\text{StartedBy}(j, i)$ iff $\sigma(i) \subset \sigma(j)$, there is no $t_j \in \sigma(j)$ such that $t_j < \sigma(i)$, but there is some $t_j \in \sigma(j)$ such that $\sigma(i) < t_j$; $\text{Ends}(i, j)$ or $\text{EndedBy}(j, i)$ iff $\sigma(i) \subset \sigma(j)$, there is no $t_j \in \sigma(j)$ such that $\sigma(i) < t_j$, but there is some $t_j \in \sigma(j)$ such that $t_j < \sigma(i)$; $\text{Holds}(p, i)$ iff $\sigma(i) \subseteq v(p)$; and $\text{Occurs}(p, i)$ iff $\sigma(i) \cap v(p) \neq \emptyset$ iff $\neg \text{Holds}(\neg p, i)$. Satisfaction is defined as usual on boolean combinations of ATL formulae. We use the notation $\mathcal{M} \models_{\text{ATL}} \varphi$ to denote the fact that the interval structure \mathcal{M} satisfies the ATL formula φ .

Therefore, $\text{Holds}(p, i)$ is satisfied iff p holds at any time point in i , while $\text{Occurs}(p, i)$ is satisfied iff p holds at some time point in i . The propositions p used in Holds and Occurs may hold at various random timepoints, so they cannot be replaced by intervals. The NP-completeness of the satisfiability problem for ATL without Holds [264] gives us immediately the NP-hardness of our ATL with Holds above. We will show in the next section that it is actually NP-complete.

In many practical applications of interest, time elapses at a discrete and enumerable rate. We next define a variant of Allen temporal algebra in which the support of the interval models are ω -sequences of time points, that

is, linear (infinite) sequences $t_1 < t_2 < t_3 < \dots < t_n < \dots$. We write these strict total orders compactly as $t_1 t_2 t_3 \dots t_n \dots$. We call the new logic **Allen Linear Temporal Logic** (ALTL). Note that ALTL has the same syntax as ATL and its satisfaction relation is defined like in ATL, but that its models are structures of the form $\mathcal{M} = (t_1 t_2 \dots, v, \sigma)$, where $t_1 t_2 \dots$ are ω -sequences of time points and σ maps intervals in \mathcal{I} into non-empty convex sets $\sigma(i)$ of $T = \{t_1, t_2, \dots\}$ (with the expected strict total ordering $<$ defined as $t_m < t_n$ iff $m < n$). It is easy to see that the convex sets of T are either finite sets of the form $\{t_m, t_{m+1}, \dots, t_n\}$ for some $0 < m \leq n$, or infinite sets of the form $\{t_m, t_{m+1}, \dots\}$ for some $0 < m$.

14.3 Linear Translation of ALTL into LTL

We next define an automatic encoding of ALTL into LTL_\square . Note that the models of ALTL differ from those of LTL in that they contain a concrete interpretation for each interval. Therefore, in order to establish a semantic relationship between the models of the two logics, we need to first add syntactic support for “intervals” to LTL . A simple way to do this is to add an atomic propositional symbol \in_i to the syntax of LTL for each interval $i \in \mathcal{I}$, with the intuition that a time point is in the interval i in a model of ALTL if and only if the proposition \in_i holds in that time point in the corresponding model of LTL . Moreover, we need to also capture, via corresponding LTL formulae, the fact that intervals are interpreted into non-empty convex sets in ALTL models.

Definition 60 Let $\mathcal{P}_{\mathcal{I}}$ be the set of atomic propositions $\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\}$ and let $\Psi_{\mathcal{I}}$ be the set of LTL formulae $\{\psi_i \mid i \in \mathcal{I}\}$ over propositions in $\mathcal{P}_{\mathcal{I}}$, where ψ_i is the formula $\Diamond \in_i \wedge \neg \Diamond(\in_i \wedge \Diamond(\neg \in_i \wedge \Diamond \in_i))$ for each $i \in \mathcal{I}$.

The following establishes the relationship between models of ALTL and of LTL :

Proposition 21 There is a bijection between $(\mathcal{P}, \mathcal{I})$ -interval models and models of $LTL(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$ that satisfy $\Psi_{\mathcal{I}}$.

Proof: Let $\mathcal{M} = (T, <, v, \sigma)$ be a tuple where $(T, <)$ is an ω -sequence, v is a map $\mathcal{P} \rightarrow 2^T$, and σ is a map $\mathcal{I} \rightarrow 2^T$; what \mathcal{M} is missing to be a model of $ALTL(\mathcal{P}, \mathcal{I})$ is the requirements that $\sigma(i)$ is non-empty and convex for any $i \in \mathcal{I}$. Then we can build a model $\mathcal{N} = (T, <, u)$ of $LTL(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}\})$,

where $u(p) = v(p)$ for all $p \in \mathcal{P}$ and $u(\epsilon_i) = \sigma(i)$ for all $i \in \mathcal{I}$. Conversely, for any model $\mathcal{N} = (T, <, u)$ of $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$ one can build a tuple $\mathcal{M} = (T, <, v, \sigma)$, where v is the restriction of u to \mathcal{P} and $\sigma(i)$ is defined as $u(\epsilon_i)$ for any $i \in \mathcal{I}$. What is left to prove is that $\sigma(i)$ is non-empty and convex for any $i \in \mathcal{I}$ if and only if $\mathcal{N} \models_{LTL} \Psi_{\mathcal{I}}$. First, note that, for any $i \in \mathcal{I}$, $\sigma(i) \neq \emptyset$ is equivalent to $\mathcal{N} \models_{LTL} \Diamond \epsilon_i$. Second, since $\sigma(i)$ is convex if and only if there are no time points t_m, t_n, t_k with $0 < m < n < k$ such that $t_m, t_k \in \sigma(i)$ and $t_n \notin \sigma(i)$, one deduces that $\sigma(i)$ is convex if and only if $\mathcal{N} \models_{LTL} \neg \Diamond(\epsilon_i \wedge \Diamond(\neg \epsilon_i \wedge \Diamond \epsilon_i))$. Therefore, $\sigma(i)$ is non-empty and convex for each $i \in \mathcal{I}$ if and only if $\mathcal{N} \models_{LTL} \Psi_{\mathcal{I}}$. \square \square

Definition 61 We let $[\cdot]$ define the bijection above, that is, if \mathcal{M} is a $(\mathcal{P}, \mathcal{I})$ -interval model then $[\mathcal{M}]$ is the corresponding model of $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$ satisfying $\Psi_{\mathcal{I}}$, defined as in the proof of Proposition 21.

We are now ready to define the first part of our syntactic encoding of ALTL formulae into LTL formulae.

Definition 62 Let $[\cdot]$ be the function taking formulae φ in $ALTL(\mathcal{P}, \mathcal{I})$ into formulae $[\varphi]$ in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$ defined inductively as follows: $[\neg \varphi]$ is $\neg[\varphi]$; $[\varphi_1 \wedge \varphi_2]$ is $[\varphi_1] \wedge [\varphi_2]$; $[Equals(i, j)]$ is $\Box(\epsilon_i \Leftrightarrow \epsilon_j)$; $[Before(i, j)]$ and $[After(j, i)]$ are $\Diamond(\epsilon_i \wedge \Diamond(\neg \epsilon_i \wedge \neg \epsilon_j \wedge \Diamond \epsilon_j))$; $[Meets(i, j)]$ and $[MetBy(j, i)]$ are $\Diamond(\epsilon_i \wedge \Diamond \epsilon_j \wedge \neg \Diamond(\epsilon_i \wedge \epsilon_j) \wedge \neg \Diamond(\neg \epsilon_i \wedge \neg \epsilon_j \wedge \Diamond \epsilon_j))$; $[Overlaps(i, j)]$ and $[OverlappedBy(j, i)]$ are $\Diamond(\epsilon_i \wedge \neg \epsilon_j \wedge \Diamond(\epsilon_i \wedge \epsilon_j \wedge \Diamond(\neg \epsilon_i \wedge \epsilon_j)))$; $[Contains(i, j)]$ and $[During(j, i)]$ are $\Diamond(\epsilon_i \wedge \neg \epsilon_j \wedge \Diamond(\epsilon_i \wedge \epsilon_j \wedge \Diamond(\epsilon_i \wedge \neg \epsilon_j)))$; $[Starts(i, j)]$ and $[StartedBy(j, i)]$ are $\Box(\epsilon_i \Rightarrow \epsilon_j) \wedge \neg \Diamond(\epsilon_j \wedge \neg \epsilon_i \wedge \Diamond \epsilon_i) \wedge \Diamond(\epsilon_j \wedge \neg \epsilon_i)$; $[Ends(i, j)]$ and $[EndedBy(j, i)]$ are $\Box(\epsilon_i \Rightarrow \epsilon_j) \wedge \Diamond(\epsilon_j \wedge \neg \epsilon_i) \wedge \neg \Diamond(\epsilon_j \wedge \epsilon_i \wedge \Diamond(\epsilon_j \wedge \neg \epsilon_i))$; $[Holds(p, i)]$ is $\Box(\epsilon_i \Rightarrow p)$; and $[Occurs(p, i)]$ is $[\neg Holds(\neg p, i)]$, that is, $\Diamond(\epsilon_i \wedge p)$.

Example 18 Let us consider again the subformula

$Meets(N_{hb}, G_b) \wedge Meets(G_b, H_b) \wedge During(G_b, @(\text{tree})) \wedge During(G_b, H)$ of the formula that characterizes the compatibilities of the monkey/banana problem (see Example 17), to illustrate how to encode an ALTL formula into an equivalent LTL $_{\Box}$ one. Its encoding is:

$$\begin{aligned} & \Diamond(\epsilon_{N_{hb}} \wedge \Diamond \epsilon_{G_b} \wedge \neg \Diamond(\epsilon_{N_{hb}} \wedge \epsilon_{G_b}) \wedge \neg \Diamond(\neg \epsilon_{N_{hb}} \wedge \neg \epsilon_{G_b} \wedge \Diamond \epsilon_{G_b})) \wedge \\ & \Diamond(\epsilon_{G_b} \wedge \Diamond \epsilon_{H_b} \wedge \neg \Diamond(\epsilon_{G_b} \wedge \epsilon_{H_b}) \wedge \neg \Diamond(\neg \epsilon_{G_b} \wedge \neg \epsilon_{H_b} \wedge \Diamond \epsilon_{H_b})) \wedge \\ & \Diamond(\epsilon_{@(\text{tree})} \wedge \neg \epsilon_{G_b} \wedge \Diamond(\epsilon_{@(\text{tree})} \wedge \epsilon_{G_b} \wedge \Diamond(\epsilon_{@(\text{tree})} \wedge \neg \epsilon_{G_b}))) \wedge \\ & \Diamond(\epsilon_H \wedge \neg \epsilon_{G_b} \wedge \Diamond(\epsilon_H \wedge \epsilon_{G_b} \wedge \Diamond(\epsilon_H \wedge \neg \epsilon_{G_b}))) \wedge (\bigwedge_{i \in \mathcal{I}} \psi_i), \end{aligned}$$

where $\mathcal{I} = \{N_{hb}, H_b, H, G_b, @(\text{tree})\}$ and ψ_i is $\Diamond \in_i \wedge \neg \Diamond (\in_i \wedge \Diamond (\neg \in_i \wedge \Diamond \in_i))$. As expected, the LTL encoding of the entire formula in Example 17 is very large. \square

The companion report [216] shows a rewriting implementation of this encoding.

Consider including the material in [216]

Theorem 21 *Given an $ALTL(\mathcal{P}, \mathcal{I})$ formula φ and a $(\mathcal{P}, \mathcal{I})$ -interval model \mathcal{M} , then $\mathcal{M} \models_{ALTL} \varphi$ iff $[\mathcal{M}] \models_{LTL} [\varphi]$.*

Proof: Structural induction on φ . If φ has the form $\neg \varphi_1$ then $\mathcal{M} \models_{ALTL} \varphi$ is equivalent to saying that it is *not* the case that $\mathcal{M} \models_{ALTL} \varphi_1$, which, by the induction hypothesis and Definition 62, is equivalent to saying that $[\mathcal{M}] \models_{LTL} [\varphi]$. The case where φ has the form $\varphi_1 \wedge \varphi_2$ is similar. What is left to show is that the property holds when φ is any of the interval relations. Let us discuss only one of them, for example $Meets(i, j)$. Suppose that $\mathcal{M} = (T, <, v, \sigma)$ and recall that $\sigma(i)$ is non-empty for any interval i . By Definition 59, $\mathcal{M} \models_{ALTL} Meets(i, j)$ iff $\sigma(i) < \sigma(j)$ and there is so $t \in T$ such that $\sigma(i) < t < \sigma(j)$. By the way $[\mathcal{M}]$ is built and because ψ_i and ψ_j ensure the non-emptiness and the convexity of the trace fragments in which \in_i and \in_j hold, This is equivalent to saying that \in_j holds *strictly* after \in_i , i.e., the $\Diamond(\in_i \wedge \Diamond \in_j \wedge \neg \Diamond(\in_i \wedge \in_j) \wedge \dots)$; part of $[Meets(i, j)]$, and that there is no period of time following \in_i that appears before \in_j in which neither \in_i nor \in_j holds, i.e., the $\Diamond(\dots \neg \Diamond(\neg \in_i \wedge \neg \in_j \wedge \Diamond \in_j))$ part of $[Meets(i, j)]$. The result can be proved similarly for the other intervals. \square

Our goal next is to reduce the satisfiability problem for $ALTL$ to LTL_{\square} satisfiability, known to be an NP-complete problem [248]. Theorem 21 gives us only half of the result, namely that if a formula φ is satisfiable in $ALTL$ then the formula $[\varphi]$ is satisfiable in LTL_{\square} . To get the other half, one could define a slightly different translation of $ALTL$ formulae, namely one that would also include the conjunction of the formulae in $\Psi_{\mathcal{I}}$. The problem with that is, however, that \mathcal{I} can be infinite, meaning that the generated LTL formula would be infinite. Fortunately, only the intervals that explicitly appear in φ need to be taken into account, thus making our transformation finite:

Definition 63 *For an $ALTL(\mathcal{P}, \mathcal{I})$ formula φ , let \mathcal{I}_{φ} be the finite set of intervals appearing in φ and let $\langle \varphi \rangle$ be the formula $[\varphi] \wedge \bigwedge \Psi_{\mathcal{I}_{\varphi}}$ in $LTL(\mathcal{P} \cup \{\in_i \mid i \in \mathcal{I}_{\varphi}\})$.*

Corollary 2 *Given a formula φ in $ALTL(\mathcal{P}, \mathcal{I})$, the following are equivalent: (1) φ is satisfiable in $ALTL(\mathcal{P}, \mathcal{I})$; (2) $\langle \varphi \rangle$ is satisfiable in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}_\varphi\})$; and (3) $\langle \varphi \rangle$ is satisfiable in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$.*

Proof: Since a model over more atomic propositions can be also regarded as a model over fewer propositions, it is immediate that 3. implies 2.. By Theorem 21, any model of φ in $ALTL(\mathcal{P}, \mathcal{I})$ yields a model of $\langle \varphi \rangle$ in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$ that satisfies $\Psi_{\mathcal{I}}$. Therefore, 1. implies 3.. To show that 2. implies 1., by Proposition 21 it suffices to show that any model in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}_\varphi\})$ satisfying $\Psi_{\mathcal{I}_\varphi}$ can be extended, by just adding appropriate valuations for the additional atomic propositions to assure that the satisfaction of φ is not affected, to a model in $LTL(\mathcal{P} \cup \{\epsilon_i \mid i \in \mathcal{I}\})$ satisfying $\Phi_{\mathcal{I}}$. This can be done many different ways. One straightforward model extension is to require that each proposition in $\{\epsilon_i \mid i \in \mathcal{I} - \mathcal{I}_\varphi\}$ holds in precisely one (arbitrary) time point. \square

Corollary 3 *The satisfiability problem for $ALTL$ is NP-complete.*

Proof: By Corollary 2, an $ALTL$ formula φ is satisfiable iff $\langle \varphi \rangle$ is satisfiable as an LTL formula. Since $\langle \varphi \rangle$ can be generated linearly in the size of the φ and since LTL -satisfiability is NP-complete, $ALTL$ -satisfiability is also NP-complete. \square

14.4 Monitoring $ALTL$

It is known that *any* monitoring algorithm for LTL -formulae requires space exponential in the size of the monitored formula [217] in the worst case. Can we find better monitoring algorithms for $ALTL$? We first argue empirically that a blind use of monitoring algorithms for LTL may be unfeasible in large applications and then propose an $ALTL$ -specific monitoring algorithm which avoids the exponential-space complexity of monitoring LTL -formulae. More precisely, we give a monitoring algorithm for $ALTL$ which only requires space (it needs to store its current state only) that is linear in the size of the input formula and whose most expensive task is to check the satisfiability of a *boolean* formula that is incrementally smaller (in the sense that some of its variables are irreversibly replaced by true or false) with each event received from the monitored system, and which initially has precisely the size of the original $ALTL$ formula.

Let us first describe informally the “monitoring problem” for a logic whose models are (finite or infinite) traces. Given a formula ξ of size n and a “running system” abstracted by its incrementally emitted events (or abstract states encoded by the atomic propositions that “hold” in them) t_1, t_2, \dots , the problem is to report when a bad prefix is reached, that is, when a finite trace $t_1 t_2 \dots t_m$ is encountered such that there is no infinite trace $t_1 t_2 \dots t_m t_{m+1} \dots$ that satisfies ξ . We here assume that storing the events is *not* an option, because their number can grow arbitrarily large. Indeed, m can be large enough so that even an algorithm that is linear in the continuously increasing execution trace at each emitted event (e.g., one that traverses the trace backwards, like the one in [217]) can become easily more impractical than one just exponential in the formula but constant in the trace (e.g., when one generates an automata monitor from it, like in [82]). One can (non-trivially) formalize the monitoring problem for a logic as a decision problem, but this is rather intricate and beyond our scope here. Here we limit ourselves to the informal problem description above and conclude that ALTL-monitoring is asymptotically as expensive as ALTL-satisfiability:

- (a) in any logic, monitoring is a harder problem than satisfiability;
- (b) for any ALTL-formula ξ , we give a monitoring algorithm which is not more expensive than checking the satisfiability of ξ .

One can readily see that monitoring is harder than satisfiability: a monitor for ξ reports violation on the empty trace iff ξ is not satisfiable. Since ALTL-satisfiability is NP-complete (Corollary 3), any monitoring algorithm for ALTL is expected to be worst-case exponential in practice. However, as in many other similar situations, this does not necessarily mean that the problem of monitoring ALTL formulae is not practical. We next briefly discuss an immediate algorithm based on the translation to *LTL*, and then give an algorithm specific to ALTL that avoids the complexity of monitoring *LTL* and which seems quite efficient in practice. The next section discusses an experiment where the ALTL formula is large enough that the *LTL*-based monitoring algorithms cannot handle it.

The transformation in Section 14.3 suggests using a general purpose monitoring algorithm for *LTL* (e.g., the one in [82]), to monitor the *LTL*

formula obtained linearly from the ALTL formula. We have experimented with this technique and have succeeded to generate, unfortunately huge, *LTL* monitors only for relatively small ALTL formulae. For example, for the ALTL formula in Example 18, which is a subformula of the ALTL formula in Example 17, the generated monitor had more than 60,000 edges, while the algorithm ran out of memory trying to generate an *LTL* monitor for the entire ALTL formula in Example 17; and that is just a toy example. The reason for our failure to generate monitors following this approach is the intermediate Büchi automata generator from *LTL* formulae; the *LTL* monitors in [82] are obtained pruning the corresponding Büchi automata (which can be exponential), by removing portions of them related to liveness – only the safety fragment of a formula is monitorable. The interested reader is encouraged to check [216] for more details on this unsuccessful approach.

We next give a monitoring algorithm for ALTL *not* based on general monitoring algorithms for *LTL*. The idea is to regard the ALTL formula φ as a *boolean proposition* in which the interval relations are regarded as special “dynamic” variables. For each interval relation we generate a little state machine, which has two special states, **true** and **false**. These state machines are shown in Figure 14.2. We also add a top-level conjunct consisting of precisely one special variable for each interval that appears in φ ; these latter variables correspond, intuitively, to the formulae ψ_i in Definition 60. The monitoring algorithm works as follows: (1) generate all the state machines in Figure 14.2 (left-top state is initial); (2) let ξ be the boolean proposition obtained from φ as above; (3) run a *boolean* satisfiability checker on ξ and stop with “error” if ξ not satisfiable; (4) otherwise, for the next event t received from the monitored system, run all the state machines one step according to t (take that deterministic edge which is satisfied by t); (5) modify the formula ξ by replacing each variable whose corresponding state machine is in a state **true** or **false** by the corresponding truth value; (6) goto step (3).

Let us briefly discuss the state machines. The ones for ψ_i ensure that intervals are contiguous (convex); some intervals can be unbounded. The next seven state machines correspond to the relations on intervals. Let us discuss the one for $Meets(i, j)$. One starts with the initial state $\widehat{(i, j)}$ (neither in i nor in j), and there it stays as far as one does not enter any of the intervals. If while in this state the monitored program enters the interval j , that is, if \in_j holds, then the relation $Meets(i, j)$ is obviously violated (interval i cannot be empty). Otherwise, if the interval i but not j is entered,

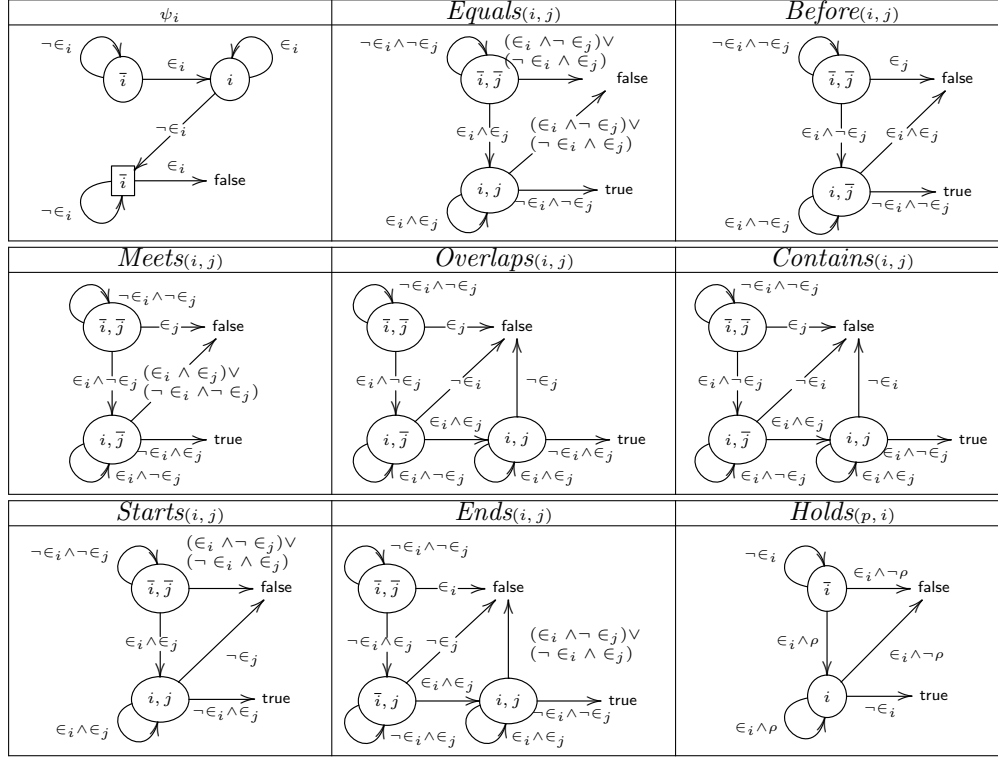


Figure 14.2: State machines are run synchronously by the monitor with each event.

then the machine moves to state (i, \bar{j}) where it waits until either i is left and j is entered in which case it returns **true**, or otherwise until i is left without entering j or i and j overlap, when it returns **false**. The machine for $Holds(p, i)$ checks that p holds during the interval i .

Example 19 Let us consider again the monkey/banana formula in Example 18,

$(Meets(N_{hb}, G_b) \wedge Meets(G_b, H_b) \wedge During(G_b, @(\text{tree})) \wedge During(G_b, H))$, and consider an execution trace which starts with the abstract events $t_1 = \{\in N_b\}$, $t_2 = \{\in N_b, \in @(\text{tree})\}$, $t_3 = \{\in G_b, \in @(\text{tree}), \in H\}$, $t_4 = \{\in H_b, \in H\}$, ..., where an abstract event formed of a set of atomic propositions is an event in which all those, and only those propositions hold. This execution trace

corresponds to the “flying monkey” scenario at the end of Example 17.

Let us simulate the execution of the ALTL monitoring algorithm above on this example. There are nine state machines like in Figure 14.2 necessary, four corresponding to each of the four interval relations and five corresponding to each interval appearing in the formula. The boolean formula ξ is just a conjunction of the corresponding nine variables. All one needs to do is to run the nine state machines on the execution trace, update the boolean proposition and then check for satisfiability after each event. After the first three events, the five ψ_i formulae will be in some intermediate (not false) states, and the four machines corresponding to the interval relations will be in the states *true*, (G_b, \overline{H}_b) , $(@(\text{tree}), G_b)$, and (G_b, H) , respectively, so the formula is still satisfiable. However, when the event t_4 is processed, the machine corresponding to $\text{During}(G_b, @(\text{tree}))$, or to $\text{Contains}(@(\text{tree}), G_b)$, transits to *false*, invalidating the boolean proposition. \square

Example 20 Consider now the ALTL formula $\neg \text{Before}(i, j)$ and a two-event trace $\{\in_i\}\{\}$. The monitoring algorithm above sets the machine corresponding to $\text{Before}(i, j)$ to state (i, \overline{j}) after processing $\{\in_i\}$ and then to state *true* after processing $\{\}$, causing the monitor to report “error” before any event containing \in_j is seen. Note that $\{\in_i\}\{\}$ is indeed a bad prefix for $\neg \text{Before}(i, j)$ (\in_j must hold eventually in any interval model of ALTL). Therefore, our monitoring algorithm for ALTL detects bad prefixes as soon as they appear. \square

Note that the state machines corresponding to ψ_i will intercept any violation of the convexity of intervals. If any of the convexities of intervals is violated, that is, if an interval starts, then it is interrupted and then started again, then the monitoring algorithm above returns “error”, because the observed trace cannot even be continued into an interval model; one can easily modify the algorithm to return a different type of error in such situations. Note also that these state machines for ψ_i do not have a *true* state: there is no way to decide by means of monitoring that ψ_i holds, because this is a property of an infinite trace; by monitoring, one can only detect the safety fragment of the inherent ALTL property “intervals are non-empty and convex”, namely the break of their convexity. Therefore, the formulae ψ_i can only detect violations of the monitored formula: their corresponding variables can only be transformed into *false*, never into *true*. If in a particular application there are external factors implying the well-formedness of intervals, then one can drop the variables (and the machines)

corresponding to ψ_i (and thus be able to also detect formula validations online).

Theorem 22 *The monitoring algorithm for ALTL above is correct.*

Proof: Thanks to the machines corresponding to ψ_i , we can assume the well-formedness of intervals in the proof. Consider some finite trace $\tau = t_1 t_2 \dots t_m$ that is well-formed wrt intervals, i.e., it can be the prefix of some interval model of ALTL. Let us first prove that for any interval relation, its corresponding state machine is in state **false** after processing τ iff τ is a bad prefix of that interval relation. We only show it for one relation, say $Before(i, j)$; the others are similar. Note that τ is a bad prefix of $Before(i, j)$ iff τ contains (some event satisfying) \in_j before or at the same time with \in_i . Since the state machine of $Before(i, j)$ reaches the state **false** iff \in_j is seen before \in_i or if \in_j and \in_i are seen together as part of an event, and since the machines corresponding to ψ_i ensure the contiguity of intervals, we can conclude that τ is a bad prefix of $Before(i, j)$ iff the corresponding machine of $Before(i, j)$ is in state **false** after processing τ .

Let us next prove that for any interval relation, the corresponding machine is in state **true** after processing τ iff τ is a good prefix of that relation, in the sense that for any infinite trace ϕ such that $\tau\phi$ is an interval model of ALTL, it is the case that $\tau\phi$ satisfies that relation. As above, let us just prove it for $Before(i, j)$. Note that the machine of $Before(i, j)$ can be in state **true** after processing τ iff τ contains no event satisfying \in_j and contains some event satisfying \in_i followed by one which does not satisfy \in_i . This is equivalent to saying that any interval model of the form $\tau\phi$ (recall that intervals have non-empty interpretations in interval models) satisfies $Before(i, j)$.

Let us now consider any ALTL formula φ and a finite trace τ as above such that the ξ formula maintained by the algorithm is satisfiable after processing τ . If φ has the form $\varphi_1 \wedge \varphi_2$ then τ is a bad (good) prefix of φ iff it is a bad (good) prefix of φ_1 or (and) φ_2 . If φ has the form $\neg\varphi_1$ then τ is a bad (good) prefix of φ iff it is a good (bad) prefix of φ_1 . Therefore, in order to test whether τ is a bad prefix of φ one only needs to know whether it is a bad prefix of φ 's interval relations, that is, if their corresponding state machines are in their corresponding **false** or **true** states after processing τ . The satisfiability checking of ξ after each event ensures that violations are reported as early as possible. \square

If one is not interested in reporting ALTL property violations as early as possible, then one can run the satisfiability checker less frequently, say once

every 100 events, or even just once at the end of the monitoring session, and thus significantly reduce the runtime overhead. If minimal runtime overhead is highly desirable, since the formula ξ to check for satisfiability changes incrementally by irreversibly transforming some of its variables into **true** or **false**, to achieve a minimal runtime overhead one can use an incremental SAT solver.

14.5 Experiment

Implementation. We have implemented a prototype monitor generation tool, called **ALTL2Monitor**. It implements the monitoring algorithm presented in the previous section using the SAT solver **zChaff** [204] for satisfiability checking.

Case Study. Our case study is a simplified version of an exploration rover (Gromit, at Nasa Ames). The mission of the robot is to visit a number of waypoints, into an initially unknown rough environment, while monitoring interesting targets on its path. The robot continuously takes pictures of the terrain in front of it, performs a stereo correlation to extract a cloud of 3D points, merges these points in its model of environment and starts this process again. In parallel, it continuously considers its current position, the next waypoint to visit, the obstacles in the model of the environment built and produces a trajectory. These two interdependent cyclic processes are synchronized. Last, a third process interrupts whenever an interesting rock has been detected. The functional layer of Gromit is implemented using functional modules (for more details see [178]). For each of them we shall consider the “visible” state variables of interest:

- **RFLEX** is the module interfaced with the low-level speed controller. It has a state variable for the position of the robot, each interval representing a specific robot position, and another one for the speed passed to the wheels controller.
- **CAMERA** shoots a pair of stereo calibrated images and saves them. It has one state variable representing the camera status (taking picture, or idle).
- **SCORREL** produces and stores a stereo correlated image. It has a state variable representing the **SCORREL** process (performing stereo correlation, or idle).
- **LANE** builds a model of the environment by aggregating clouds of 3D points produced by **SCORREL**. It services two requests: read in an internal buffer and fuse the read. **LANE** has one state variable for the model building

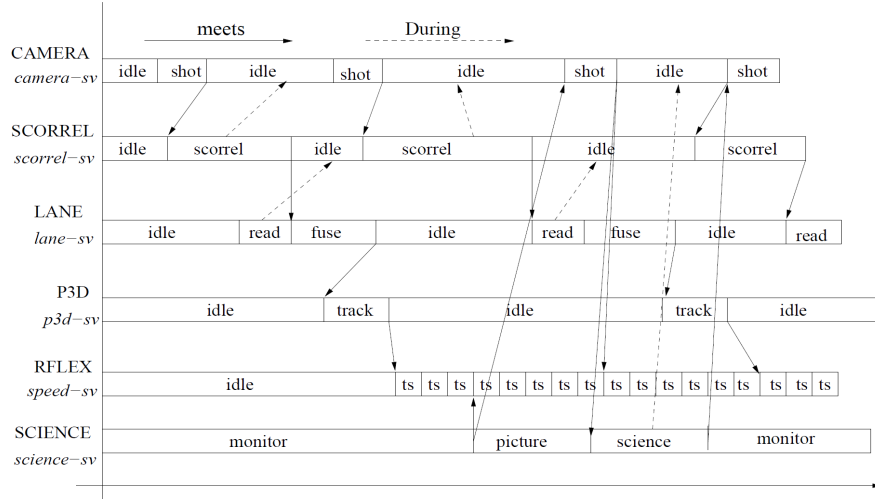


Figure 14.3: Partial Gromit Model: Attributes and compatibilities

process.

- **P3D** is a rover navigation software. It produces an arc trajectory which is translated in a speed reference, to try to reach a waypoint. P3D has a variable for its state (*idle* or computing the speed) and one for the waypoints to visit.
- **SCIENCE**. This module monitors a particular condition of interest to scientist (such as detecting a rock with particular features). When such a condition arises while the robot is moving toward a waypoint, it stops and takes a picture of the rock. It has one state variable for its state (monitoring interesting rock or *idle*).

Figure 14.3 shows some temporal relations representing a simplified version of the actual Gromit Rover.

Results. Due to intellectual property restrictions, we did not have access to the execution platform of the Gromit Rover. However, the CNRS Laboratory LAAS (at Toulouse, France) provided¹ us with a file formalizing some of the compatibilities as an ATL formula of more than 100 interval relations, as well as with a set of one hundred traces generated by Gromit Rover execution platform. We applied our prototype ALTL2Monitor off-line to check these traces; the checking took negligible time. However, the satisfiability checker

¹We warmly thank Felix Ingrand for help.

was applied only once at the end of the monitoring session of each trace, because we expected the traces to be correct, which was indeed the case.

14.6 Conclusion

We presented Allen linear temporal logic (ALTL), an automated translation of ALTL into LTL, a monitor synthesis algorithm for ALTL, as well as a real-life experiment. While *LTL* can be a suitable logic for AI and planning, we also believe that ALTL can be a suitable logic for certain program verification efforts. Its simplicity, neutrality and visual interpretation cannot be ignored. We plan to apply our ALTL monitoring prototype to the autonomous embedded system iRobot ATRV of the LAAS Laboratory.

Exercises

Exercise 28 *Definition 62 shows how to translate ALTL to future-time LTL. Give an equivalent translation of ALTL to past-time LTL, and exemplify it on the ALTL formula in Example 18. Compare monitoring ALTL using the algorithm described in Section 14.4, with monitoring the corresponding past-time LTL using the optimized algorithm in Chapter 10.*

Chapter 15

Parametric Property Monitoring

Material from [228]

Abstract: Analysis of execution traces plays a fundamental role in many program analysis approaches, such as runtime verification, testing, monitoring, and specification mining. Execution traces are frequently parametric, i.e., they contain events with parameter bindings. Each parametric trace usually consists of many meaningful *trace slices* merged together, each slice corresponding to one parameter binding. For example, a Java program creating iterator objects i_1 and i_2 over collection object c_1 may yield a trace $\text{createIter}\langle c_1\ i_1 \rangle\ \text{next}\langle i_1 \rangle\ \text{createIter}\langle c_1\ i_2 \rangle\ \text{updateColl}\langle c_1 \rangle\ \text{next}\langle i_1 \rangle$ parametric in collection c and iterator i , whose slices corresponding to instances “ $c, i \mapsto c_1, i_1$ ” and “ $c, i \mapsto c_1, i_2$ ” are $\text{createIter}\langle c_1\ i_1 \rangle\ \text{next}\langle i_1 \rangle\ \text{updateColl}\langle c_1 \rangle\ \text{next}\langle i_1 \rangle$ and, respectively, $\text{createIter}\langle c_1\ i_2 \rangle\ \text{updateColl}\langle c_1 \rangle$. Several approaches have been proposed to specify and dynamically analyze parametric properties, but they have limitations: some in the specification formalism, others in the type of trace they support. Not unexpectedly, the existing approaches share common notions, intuitions, and even techniques and algorithms, suggesting that a fundamental study and understanding of parametric trace analysis is necessary.

This foundational paper aims at giving a semantics-based solution to parametric trace analysis that is unrestricted by the type of parametric property or trace that can be analyzed. Our approach is based on a rigorous understanding of *what* a parametric trace/property/monitor is and *how* it relates to its non-parametric counter-part. A general-purpose parametric

trace slicing technique is introduced, which takes each event in the parametric trace and dispatches it to its corresponding trace slices. This parametric trace slicing technique can be used in combination with any conventional, non-parametric trace analysis technique, by applying the later on each trace slice. As an instance, a parametric property monitoring technique is then presented, which processes each trace slice online. Thanks to the generality of parametric trace slicing, the parametric property monitoring technique reduces to encapsulating and indexing unrestricted and well-understood non-parametric property monitors (e.g., finite or push-down automata).

The presented parametric trace slicing and monitoring techniques have been implemented and extensively evaluated. Measurements of runtime overhead confirm that the generality of the discussed techniques does not come at a performance expense when compared with existing parametric trace monitoring systems.

15.1 Introduction and Motivation

Parametric traces, i.e., traces containing events with parameter bindings, abound in program executions, because they naturally appear whenever abstract parameters (e.g., variable names) are bound to concrete data (e.g., heap objects) at runtime. In this section we first discuss some motivating examples and describe the problem addressed in this paper, then we recall related work and put the work in this paper in context, and then we explain our contributions and finally the structure of the paper.

15.1.1 Motivating examples and highlights

We here describe three examples of parametric properties, in increasing difficulty order, and use them to highlight and motivate the semantic results and the algorithms presented in the rest of the paper.

Typestates [260] refine the notion of type by stating not only what operations are allowed by a particular object, but also what operations are allowed in what contexts. Typestates are particular parametric properties with only one parameter. Figure 17.1 shows the typestate description for a property saying that it is invalid to call the `next()` method on an iterator object when there are no more elements in the underlying collection, i.e., when `hasnext()` returns false, or when it is unknown if there are more elements in the collection, i.e., `hasnext()` is not called. From the *unknown* state, it is

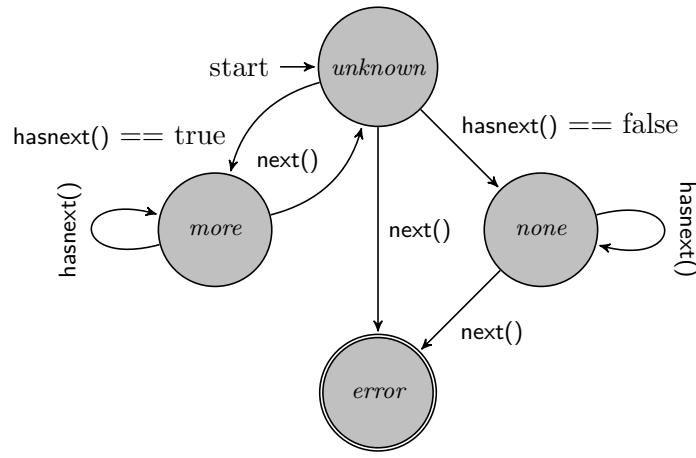


Figure 15.1: Typestate property describing the correct use of the `next()` and `hasnext()` methods.

always an error to call the `next()` method because such an operation could be unsafe. If `hasnext()` is called and returns `true`, it is safe to call `next()`, so the typestate enters the *more* state. If, however, the `hasnext()` method returns `false`, there are no more elements, and the typestate enters the *none* state. In the *more* and *none* states, calling the `hasnext()` method provides no new information. It is safe to call `next()` from the *more* state, but it becomes *unknown* if more elements exist, so the typestate reenters the initial *unknown* state. Finally, calling `next()` from the *none* state results in an error. For simplicity, we here assume that `next()` is the only means to modify the state of an iterator; concurrent modifications are discussed in other examples shortly.

It is straightforward to represent the typestate property in Figure 17.1, and all typestate properties, as particular (one-parameter) *parametric* properties. Indeed, the behaviors described by the typestate in Figure 17.1 are intended to be obeyed by all iterator object instances; that is, we have a property parametric in the iterator. To make this more precise, let us look at the problem from the perspective of observable program *execution traces*. A trace can be regarded as a sequence of *events* relevant to the property of interest, in our case calls to `next()` or to `hasnext()`; the latter can be further split into two categories, one when `hasnext()` returns `true` and the other when it returns `false`. Since the individuality of each iterator matters, we must regard each event as being *parametric* in the iterator yielding it. Formally,

traces relevant to our typestate property are formed with three parametric events, namely $\text{next}\langle i \rangle$, $\text{hasnexttrue}\langle i \rangle$, and $\text{hasnextfalse}\langle i \rangle$. A possible trace can be $\text{hasnexttrue}\langle i_1 \rangle \text{hasnextfalse}\langle i_2 \rangle \text{next}\langle i_1 \rangle \text{next}\langle i_2 \rangle \dots$, which violates the typestate property for iterator instance i_2 . How to obtain execution traces is not our concern here (several runtime monitoring systems use AspectJ instrumentation). Our results in this paper are concerned with how to specify properties over parametric traces, what is their meaning, and how to monitor them.

Let us first briefly discuss our approach to specifying properties over parametric execution traces, that is, *parametric properties*. To keep it as easy as possible for the user and to leverage our knowledge on specifying ordinary, non-parametric properties, we build our specification approach on top of *any* formalism for specifying non-parametric properties. More precisely, all one has to do is to first specify the property using any conventional formalism as if there were only one possible instance of its parameters, and then use a special Λ quantifier to make it parametric. For our typestate example, suppose that typestate is the finite state machine in Figure 17.1, modified by replacing the method calls on edges with actual events as described above. Then the desired parametric property is $\Lambda i . \text{typestate}$. The meaning of this parametric property is that whatever was intended for its non-parametric counterpart, *typestate*, must hold *for each* parameter instance; that is, the *error* state must not be reached for any iterator instance, which is precisely the desired meaning of this typestate. Another way to specify the same property is using a regular expression matching all the good behaviors, each bad prefix for any instance thus signaling a violation:

$$\Lambda i . (\text{hasnexttrue}\langle i \rangle^+ \text{next}\langle i \rangle \mid \text{hasnextfalse}\langle i \rangle^*)^*$$

Yet another way to specify its non-parametric part is with a linear-temporal logic formula:

$$\Lambda i . \Box(\text{next}\langle i \rangle \implies \odot \text{hasnexttrue}\langle i \rangle)$$

The above LTL formula says “it is always (\Box) the case that each $\text{next}\langle i \rangle$ event is preceded (\odot) by a $\text{hasnexttrue}\langle i \rangle$. The LTL formula must hold for any iterator instance. In general, if $\Lambda X . P$ is a parametric property, where X is a set of one or more parameters, we may call P its corresponding *base* or *root* or *non-parametric property*. In this paper we develop a mathematical foundation for specifying such parametric properties independently of the formalism used for specifying their non-parametric part, define their precise semantics, provide algorithms for online monitoring of parametric properties,

and finally bring empirical evidence showing that monitoring parametric properties is in fact feasible.

Parametric properties properly *generalize* tpestates in two different directions. First, parametric properties allow more than one parameter, allowing us to specify not only properties about a given object such as the tpestate example above, but also properties that capture *relationships between objects*. Second, they allow us to specify infinite-state root properties using formalisms like context-free grammars (see Sections 15.2.4, 15.2.5 and 15.2.6).

Let us now consider a two-parameter property. Suppose that one is interested in analyzing collections and iterators in Java. Then execution traces of interest may contain events `createIter⟨c i⟩` (iterator i is created for collections c), `updateColl⟨c⟩` (c is modified), and `next⟨i⟩` (i is accessed using its next element method), instantiated for particular collection and iterator instances. Most properties of parametric traces are also parametric; for our example, a property may be “collections are not allowed to change while accessed through iterators”, which is parametric in a collection *and* an iterator. The parametric property above expressed as a regular expression (here matches mean violations) can be

$$\Lambda c, i. \text{ createIter}\langle c i \rangle \text{ next}\langle i \rangle^* \text{ updateColl}\langle c \rangle^+ \text{ next}\langle i \rangle$$

From here on, when we know the number and types of parameters of each event, we omit writing them in parametric properties, because they are redundant; for example, we write

$$\Lambda c, i. \text{ createIter next}^* \text{ updateColl}^+ \text{ next}$$

Parametric properties, unfortunately, are very hard to formally verify and validate against real systems, mainly because of their dynamic nature and potentially huge or even unlimited number of parameter bindings. Let us extend the above example: in Java, one may create a collection from a map and use the collection’s iterator to operate on the map’s elements. A similar safety property is: “maps are not allowed to change while accessed indirectly through iterators”. Its violation pattern is:

$$\Lambda m, c, i. \text{ createColl} (\text{updateMap} \mid \text{updateColl})^* \text{ createIter next}^* (\text{updateMap} \mid \text{updateColl})^+ \text{ next}$$

with two new parametric events `createColl⟨m c⟩` (collection c is created from map m) and `updateMap⟨m⟩` (m is updated). All the events used in this property

provide only partial parameter bindings (`createColl` binds only m and c , etc.), and parameter bindings carried by different events may be combined into larger bindings; e.g., `createColl` $\langle m_1 c_1 \rangle$ can be combined with `createter` $\langle c_1 i_1 \rangle$ into a full binding $\langle m_1 c_1 i_1 \rangle$, and also with `createter` $\langle c_1 i_2 \rangle$ into $\langle m_1 c_1 i_2 \rangle$. It is highly challenging for a trace analysis technique to correctly and efficiently maintain, locate and combine trace slices for different parameter bindings, especially when the trace is long and the number of parameter bindings is large.

This paper addresses the problem of parametric trace analysis from a foundational, semantic perspective:

Given a parametric trace τ and a parametric property $\Lambda X . P$, what does it mean for τ to be a good or a bad trace for $\Lambda X . P$? How can we show it? How can we leverage, to the parametric case, our knowledge and techniques to analyze conventional, non-parametric traces against conventional, non-parametric properties?

In this paper we first formulate and then rigorously answer and empirically validate our answer to these questions, in the context of runtime verification. In doing so, a technique for trace slicing is also presented and shown correct, which we regard as one of the central results in parametric trace analysis. In short, our overall approach to monitor a parametric property $\Lambda X . P$ is to observe the parametric trace as it is being generated by the running system, slice it online with respect to the used parameter instances, and then send each slice piece-wise to a non-parametric monitor corresponding to the base property P ; this way, multiple monitor instances for P can and typically do coexist, one for each trace slice.

The main conceptual limitation of our approach is that the parametrization of properties is only allowed at the top-level, that is, the base property P in the parametric property $\Lambda X . P$ cannot have any Λ binders. In other words, we do not consider nested parameters. To allow nested parameters one needs a syntax for properties, so that one can incorporate the syntax for parameters within the syntax for properties, likely at specific places only. However, one of our major goals is to be formalism-independent, which means that, by the nature of the problem that we are attempting to solve, we can only parameterize properties at the top. Many runtime verification approaches deliberately accept the same limitation, as discussed below, because arbitrarily nested parameters are harder to understand and turn out to generate higher runtime overhead in the systems supporting them.

Our concrete contributions are explained after the related work.

15.1.2 Related Work

We here discuss several major approaches that have been proposed so far to specify and monitor parametric properties, and relate them to our subsequent results in this paper. It is worth mentioning upfront that, except for the MOP approach [198] which motivated and inspired the work in this paper, the existing approaches do *not* follow the general methodology proposed by our approach in this paper. More precisely, they employ a monolithic monitoring approach, where one monitor is associated to each parametric property; the monitor receives and handles each parametric event in a formalism-specific way. In contrast, our approach is to generate multiple local monitors, each keeping track of one parameter instance. Our approach leads not only to a lower runtime overhead as empirically shown in Section 15.9, but it also allows us to separate concerns (i.e., the parameter handling from the specification formalism and monitor synthesis for the basic property) and thus potentially enabling a broader spectrum of optimizations that work for various different property specification formalisms and corresponding monitors.

Tracematches [9, 24] is an extension of AspectJ [210] supporting parametric regular patterns; when patterns are matched during the execution, user-defined advice can be triggered. J-LO [39] is a variation of Tracematches that supports a first-order extension variant of linear temporal logic (LTL) that supports data parametrization by means of quantifiers [257]; the user-provided actions are executed when the LTL properties are violated. Also based on AspectJ, [189] proposes Live Sequence Charts (LSC) [81] as an inter-object scenario-based specification formalism; LSC is implicitly parametric, requiring parameter bindings at runtime. Tracematches, J-LO and LSC [189] support a limited number of parameters, and each has its own approach to handle parameters, specific to its particular specification formalism. Our semantics-based approach in this paper is generic in the specification formalism and admits, in theory, a potentially unlimited number of parameters. In spite of the generality of our theoretical results, we chose in our current implementations (see Section 15.9) to also support only a bounded number of parameters, like in the aforementioned approaches.

JavaMOP [198, 62] (<http://javamop.org>) is a parametric specification and monitoring system that is generic in the specification formalism for base properties, each formalism being included as a logic plugin. Monitoring code is generated from parametric specifications and woven within the original Java program, also using AspectJ, but using a different approach that allows it to encapsulate monitors for non-parametric properties as blackboxes. Until recently, JavaMOP’s genericity came at a price: it could only monitor execution traces in which the first event in each slice instantiated all the property parameters. This limitation prevented the JavaMOP system presented in [62] from monitoring some basic parametric properties, including ones discussed in this paper. Our novel approach to parametric trace slicing and monitoring discussed in this paper does not have that limitation anymore. The parametric slicing and monitoring technique discussed in this paper has been incorporated both in JavaMOP [198] and in its commercial-grade successor RV [197], together with several optimizations that we do not discuss here; Section 15.9 discusses experiments done with both these systems, as well as with Tracematches, for comparison, because Tracematches has proven to be the most efficient runtime verification system besides JavaMOP.

Program Query Language (PQL) [192] allows the specification and monitoring of parametric context-free grammar (CFG) patterns. Unlike the approaches above that only allow a bounded number of property parameters, PQL can associate parameters with sub-patterns that can be recursively matched at runtime, yielding a potentially unbounded number of parameters. PQL’s approach to parametric monitoring is specific to its particular CFG-based specification formalism. Also, PQL’s design does not support arbitrary execution traces. For example, field updates and method begins are not observable; to circumvent the latter, PQL allows for observing traces local to method calls. Like PQL, our technique also allows an unlimited number of parameters (but as mentioned above, our current implementation supports only a bounded number of parameters). Unlike PQL, our semantics and techniques are not limited to particular events, and are generic in the property specification formalism; CFGs are just one such possible formalism.

Eagle [30], RuleR [32], and Program Trace Query Language (PTQL) [118] are very general trace specification and monitoring systems, whose specification formalisms allow complex properties with parameter bindings anywhere in the specification (not only at the beginning, like we do). Eagle and RuleR are based on fixed-point logics and rewrite rules, while PTQL is based on SQL relational queries. These systems tackle a different aspect of

generality than we do: they attempt to define general specification formalisms supporting data binding among many other features, while we attempt to define a general parameterization approach that is logic-independent. As discussed in [24, 195, 62] (Eagle and PQL cases), the very general specification formalisms tend to be slower; this is not surprising, as the more general the formalism the less the potential for optimizations. Our techniques can be used as an optimization for certain common types of properties expressible in these systems: use any of these to specify the base property P , then use our generic techniques to analyze $\Lambda X . P$.

15.1.3 Contributions

Besides proposing a formal semantics to parametric traces, properties, and monitoring, we make two theoretical contributions and discuss implementations that validate them empirically:

1. Our first result is a general-purpose online parametric trace slicing algorithm (algorithm $\mathbb{A}\langle X \rangle$ in Section 15.6) together with its proof of correctness (Theorem 23), positively answering the following question: *given a parametric execution trace, can one effectively find the slices corresponding to each parameter instance without having to traverse the trace for each instance?*
2. Our second result, building upon the slicing algorithm, is an online monitoring technique (algorithms $\mathbb{B}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ in Section 15.8) together with its proof of correctness (Theorems 24 and 25), which positively answers the following question: *is it possible to monitor arbitrary parametric properties $\Lambda X . P$ against parametric traces, provided that the root property P is monitorable using conventional monitors?*
3. Finally, our implementation of these techniques in the JavaMOP and RV systems positively answers the following question: *can we implement general purpose and unrestricted parametric property monitoring tools which are comparable in performance with or even outperform existing parametric property monitoring tools on the restricted types of properties and/or traces that the latter support?*

Preliminary results reported in this paper have been published in a less polished form as a technical report in summer 2008 [221]. Then a shorter, conference paper was presented at TACAS 2009 in York, U.K. [65]. This extended paper differs from [65] as follows:

1. It defines all the mathematical infrastructure needed to prove the results claimed in [65]. For example, Section 15.3 is new.
2. It expands the results in [65] and includes all their proofs, as well as additional results needed for those proofs. For example, Section 15.5 is new.
3. It discusses more examples of parametric properties. For example, Section 15.2 is new.
4. The implementation section in [65] presented an incipient implementation of our technique in a prototype system called PMon there (from Parametric Monitoring). In the meanwhile, we have implemented the technique described in this paper as an integral part of the runtime verification systems JavaMOP (<http://javamop.org>) and RV [197]. The implementation section (Section 15.9) now refers to these systems.

15.1.4 Paper Structure

Section 15.2 discusses examples of parametric properties. Section 15.3 provides the mathematical background needed to formalize the concepts introduced later in the paper. Section 15.4 formalizes parametric events, traces and properties, and defines trace slicing. Section 15.5 establishes a tight connection between the parameter instances in a trace and the parameter instance used for slicing. Sections 15.6, 15.7 and 15.8 discuss our main techniques for parametric trace slicing and monitoring, and prove them correct. Section 15.9 discusses implementations of these techniques in two related systems, JavaMOP and RV. Section 15.9 concludes and proposes future work.

15.2 Examples of Parametric Properties

In this section we discuss several examples of parametric properties. Our purpose here is twofold. On the one hand we give the reader additional intuition and motivation for the subsequent semantics and algorithms, and, on the other hand, we justify the generality of our approach with respect to the employed specification formalism for trace properties. The discussed examples of parametric properties are defined using various trace specification formalisms, some with more than one parameter and some with more than validating and/or violating categories of behaviors. For each of the examples,

we give hints on how our subsequent techniques in Sections 15.6 and 15.8 work. In order to explain the examples in this section we also informally introduce necessary notions, such as events and traces (both parametric and non-parametric); all these notions will be formally defined in Section 15.4.

For each example, we also discuss which of the existing runtime verification systems can support it. Note that JavaMOP [198] and its commercial-grade successor RV [197], which build upon the trace slicing and monitoring techniques presented in this paper, are the only runtime verification systems that support all the parametric properties discussed below.

15.2.1 Releasing acquired resources

Consider a certain type of resource (e.g., synchronization objects) that can be acquired and released by a given procedure, and suppose that we want the resources of this type to always be explicitly released by the procedure whenever acquired and only then. This example will be broken in subparts and used as a running example in Section 15.4 to introduce our main notions and notations.

Let us first consider the non-parametric case in which we have only one resource. Supposing that the four events of interest, i.e., the begin/end of the procedure and the acquire/release of the resource, are $\mathcal{E} = \{\text{begin}, \text{end}, \text{acquire}, \text{release}\}$, then the following regular pattern P captures the desired behavior requirements:

$$P = (\text{begin}(\epsilon \mid (\text{acquire}(\text{acquire} \mid \text{release})^* \text{release}))\text{end})^*$$

The above regular pattern states that the procedure can take place multiple times and, if the resource is acquired then it is released by the end of the procedure (ϵ is the empty word). For simplicity, we here assume that the procedure is not recursive and that the resource can be acquired and released multiple times, with the effect of acquiring and respectively releasing it precisely once; Section 15.2.4 shows how to use a context-free pattern to specify possibly recursive procedures with matched acquire/release events within each procedure invocation. One matching execution trace for this property is, e.g., `begin acquire acquire release end begin end`.

Let us now consider the parametric case in which we may have more than one resource and we want each of them to obey the requirements specified above. Now the events `acquire` and `release` are parametric in the resource being acquired or released, that is, they have the form `acquire` $\langle r_1 \rangle$, `release` $\langle r_2 \rangle$, etc.

The begin/end events take no parameters, so we write them $\text{begin}\langle\rangle$ and $\text{end}\langle\rangle$.

A parametric trace τ for our running example can be the following:

$$\tau = \text{begin}\langle\rangle \text{acquire}\langle r_1 \rangle \text{acquire}\langle r_2 \rangle \text{acquire}\langle r_1 \rangle \text{release}\langle r_1 \rangle \text{end}\langle\rangle \text{begin}\langle\rangle \text{acquire}\langle r_2 \rangle \text{release}\langle r_2 \rangle \text{end}\langle\rangle$$

This trace involves two resources, r_1 and r_2 , and it really consists of *two trace slices* merged together, one for each resource:

$$\begin{aligned} \langle r_1 \rangle : & \text{begin acquire acquire release end begin end} \\ \langle r_2 \rangle : & \text{begin acquire end begin acquire release end} \end{aligned}$$

The begin and end events belong to both trace slices. Since we know the parameter instance for each trace slice and we know the types of parameters for each event, to avoid clutter we do not mention the redundant parameter bindings of events in trace slices.

Our trace slicing algorithm discussed in Section 15.6 processes the parametric trace only once, traversing it from the first parametric event to the last, incrementally calculating a collection of meaningful trace slices so that it can quickly identify and report the slice corresponding to any parameter instance when requested.

Note that the $\langle r_1 \rangle$ trace slice matches the specification P above, while the $\langle r_2 \rangle$ trace slice does not. To distinguish parametric properties referring to multiple trace slices from ordinary properties, we explicitly list the parameters using a special Λ binder. For example, our property above parametric in the resource r is $\Lambda r . P$, or

$$\Lambda r . (\text{begin } (\epsilon \mid (\text{acquire } (\text{acquire} \mid \text{release})^* \text{release})) \text{end})^*$$

Both Tracematches [9, 24] and JavaMOP [198] can specify/monitor such parametric regular properties, the latter using its extended-regular expression (ERE) plugin.

For the sake of a terminology, P is called a non-parametric, or a root, or a basic property, in contrast to $\Lambda r . P$, which is called a parametric property. As detailed in Section 15.4, parametric properties are functions taking a parametric trace (e.g., τ) and a parameter instance (e.g., $r \mapsto r_1$ or $r \mapsto r_2$) into a verdict category for the basic property P (e.g., *match* or *fail*). In our case, the semantics of our parametric property $\Lambda r . P$ takes parametric trace τ and parameter instance $r \mapsto r_1$ to *match*, and takes τ and $r \mapsto r_2$ to *fail*, that is,

$$\begin{aligned} (\Lambda r . P)(\tau)(r \mapsto r_1) &= \text{match} \\ (\Lambda r . P)(\tau)(r \mapsto r_2) &= \text{fail} \end{aligned}$$

Our parametric monitoring algorithm in Section 15.8 reports a fail for instance $r \mapsto r_2$ precisely when the first `end` event is encountered.

We would like to make two observations at this stage. First, as we already mentioned, we only parameterize a property at the top, that is, the Λ binder cannot be used inside the basic property. Indeed, since we do not enforce any particular syntax for basic properties, it is not clear how to mix the Λ binder with the inexistent property constructs. Second, one should not confuse our parameters with universally quantified variables. While in our example above Λ may feel like a universal quantifier, note that one may prefer to specify the same parametric property in a more negative fashion, for example to specify the bad behaviors instead of the positive ones. Relying on the fact that the `begin` and `end` events must be correctly matched, one can only state the bad patterns, which are a `begin` followed by a `release` and an `acquire` followed by an `end`:

$$\Lambda r . (\mathcal{E}^*(\text{begin release} \mid \text{acquire end}) \mathcal{E}^*)$$

The right way to regard a parametric property is as one indexed by all possible instances of the parameters, each instance having its own interpretation of the trace (only caring of the events relevant to it), which is orthogonal to the other instances' interpretations.

15.2.2 Authenticate before use

Consider a server authenticating and using keys, say k_1, k_2, k_3 , etc., whose execution traces contain events `authenticate` $\langle k_1 \rangle$, `use` $\langle k_2 \rangle$, etc. A possible trace of such a system can be

$$\tau = \text{authenticate}\langle k_1 \rangle \text{authenticate}\langle k_3 \rangle \text{use}\langle k_3 \rangle \text{use}\langle k_2 \rangle \text{authenticate}\langle k_2 \rangle \text{use}\langle k_1 \rangle \text{use}\langle k_2 \rangle \text{use}\langle k_3 \rangle$$

A parametric property for such a system can be “each key must be authenticated before use”, which, using linear temporal logic (LTL) as a specification formalism for the corresponding base property, can be expressed as

$$\Lambda k . \Box(\text{use} \rightarrow \Diamond \text{authenticate})$$

Such parametric LTL properties can be expressed in both J-LO [39] and JavaMOP [198, 62] (the later using its LTL logic plugin). For the trace above, the trace slice corresponding to k_3 is `authenticate use use` corresponding to the parametric subtrace `authenticate` $\langle k_3 \rangle$ `use` $\langle k_3 \rangle$ `use` $\langle k_3 \rangle$ of events relevant to k_3 in τ , but keeping only the base events; also, the trace slice corresponding

to k_2 is `use authenticate use`. Our trace slicing algorithm in Section 15.6 can detect these slices. Moreover, with the finite trace LTL semantics in [223],

$$\begin{aligned} (\Lambda k . \Box(\text{use} \rightarrow \Diamond \text{authenticate}))(\tau)(k \mapsto k_3) &= \text{true} \\ (\Lambda k . \Box(\text{use} \rightarrow \Diamond \text{authenticate}))(\tau)(k \mapsto k_2) &= \text{false} \end{aligned}$$

Our parametric monitoring algorithm in Section 15.8 reports a violation for instance $k \mapsto k_2$ precisely when the first `use` $\langle k_2 \rangle$ is encountered.

15.2.3 Safe iterators

Consider the following property for iterators created over vectors: when an iterator is created for a vector, one is not allowed to modify the vector while its elements are traversed using the iterator. The JVM usually throws a runtime exception when this occurs, but the exception is not guaranteed in a multi-threaded environment. Supposing that parametric event `create` $\langle v\ i \rangle$ is generated when iterator i is created for vector v , `update` $\langle v \rangle$ is generated when v is modified, and `next` $\langle i \rangle$ is generated when i is accessed using its “next element” interface, then one can write it as the parametric regular property

$$\Lambda v, i . \text{create next}^* \text{update}^+ \text{next}.$$

Such parametric regular expression properties can be expressed in both Tracematches [9] and JavaMOP [198, 62] (the latter using its ERE plugin). We here assumed that the matching of the regular expression corresponds to violation of the base property. Thus, the parametric property is violated by a given trace and a given parameter instance whenever the regular pattern above is matched by the corresponding trace slice. For example, if $\tau = \text{create}\langle v_1\ i_1 \rangle \text{next}\langle i_1 \rangle \text{create}\langle v_1\ i_2 \rangle \text{update}\langle v_1 \rangle \text{next}\langle i_1 \rangle$ is a parametric trace where two iterators are created for a vector, then the slice corresponding to $\langle v_1\ i_1 \rangle$ is `create next update next` and the one corresponding to $\langle v_1\ i_2 \rangle$ is `create update`, so τ violates the parametric property (i.e., matches the regular pattern above) on instance $\langle v_1\ i_1 \rangle$, but not on instance $\langle v_1\ i_2 \rangle$. Note that in this example there are more than one parameters in events, traces and property, namely a vector and an iterator. Indeed, the main difficulty of our techniques in Sections 15.6 and 15.8 was precisely to handle general purpose parametric properties with an arbitrary number of parameters. The slicing algorithm in Section 15.6 processes parametric traces and maintains enough slicing information so that, when asked to produce slices corresponding to particular parameter instances, e.g., to $\langle v_1\ i_2 \rangle$, it can do so without any

further analysis of the trace. Also, in this case, the monitoring algorithm in Section 15.8 reports a match each time a parameter instance yields a matching trace slice.

15.2.4 Correct locking

Consider a custom implementation of synchronization in which one can acquire and release locks manually (like in Java 5 and later versions). A basic property is that each function releases each lock as many times as it acquires it. Assuming that the executing code is always inside some function (like in Java, C, etc.), that `begin⟨⟩` and `end⟨⟩` events are generated whenever function executions are started and terminated, respectively, and that `acquire⟨l⟩` and `release⟨l⟩` events are generated whenever lock l is acquired or released, respectively, then one can specify this safety property using the following parametric context-free grammar (CFG) pattern:

$$\Lambda l . S \rightarrow S \text{ begin } S \text{ end} \mid S \text{ acquire } S \text{ release} \mid \epsilon$$

Such parametric CFG properties can be expressed in both PQL [192] and JavaMOP [198, 195] (the later using its CFG plugin). We here borrow the CFG property semantics of the CFG plugin of JavaMOP (and also RV [197]) in [195], that is, this parametric property is violated by a parametric execution with a given parameter instance (i.e., concrete lock) whenever the corresponding trace slice cannot be completed into one accepted by the grammar language. While this property can be expressed in JavaMOP and even monitored in its non-parametric form, the previous implementation of JavaMOP in [195] cannot monitor it as a parametric property because its violating traces most likely start with a property-relevant `begin⟨⟩` event, which does not contain a lock parameter; therefore, the previous limitation of JavaMOP (allowing only events that instantiate all property’s parameters to create a monitor instance) did not allow us to monitor this natural CFG property. To circumvent this limitation, [195] proposed a different way to specify this property, in which the violating traces started with an `acquire⟨l⟩` event. We do not need such artificial encodings anymore in the new version of JavaMOP, and they were never needed in RV (RV improves the new JavaMOP).

For profiling reasons, one may also want to take notice of validations, or matches of the property, as well as matches followed by violation, etc.; one can therefore have different interpretations of CFG patterns as base properties, classifying traces into various categories. What is different in

this example, compared to the previous ones, is that the non-parametric property cannot be implemented as a finite state machine. With the CFG monitoring algorithm proposed in [195] used to monitor the base property, our parametric monitoring algorithm in Section 15.8 reports a violation of this parametric CFG property as soon as a parameter instance is detected for which the corresponding trace slice has no future, that is, it admits no continuation into a trace in the language of the grammar.

15.2.5 Safe resource use by safe client

A client can use a resource only within a given procedure and, when that happens, both the client and the resource must have been previously authenticated as part of that procedure. Assuming the procedure fixed and given, this is a property over traces with five types of events: begin and end of the procedure ($\text{begin}\langle\rangle$ and $\text{end}\langle\rangle$), authenticate of client ($\text{auth-client}\langle c\rangle$) or of resource ($\text{auth-resource}\langle r\rangle$), and use of resource by client ($\text{use}\langle r\ c\rangle$). Using the past time linear temporal logic with calls and returns (ptCaRet) in [229], one would write it as follows:

$$\Lambda r, c. \text{ use} \rightarrow ((\overline{\Diamond} \text{ begin}) \wedge \neg((\neg \text{auth-client}) \overline{\mathcal{S}} \text{ begin}) \wedge \neg((\neg \text{auth-resource}) \overline{\mathcal{S}} \text{ begin}))$$

The overlined operators are abstract variants of temporal operators, in the sense that they are defined on traces that collapse terminated procedure calls (erase subtraces bounded by matched begin/end events). For example, “ $\overline{\Diamond} \text{ begin}$ ” holds only within a procedure call, because all the nested and terminated procedure calls are abstracted away. In words, the above says: if one sees the use of the resource (use) then that must take place within the procedure and it is not the case that one did not see, within the main procedure since its latest invocation, the authentication of the client or the authentication of the resource.

JavaMOP can express this property using its ptCaRet logic plugin [229]. However, until recently [198], JavaMOP could again only monitor it in its non-parametric form, because of its previous limitation allowing only completely parameterized events to create monitors. Even though it may appear that this property can only be violated when a completely parameterized $\text{use}\langle r\ c\rangle$ event is observed, in fact, the monitor must already exist at that point in the execution and “know” whether the client and the resource have authenticated since the begin of the current procedure; all the other events involved in the property are incompletely parameterized, so, unfortunately, this parametric

property could not be monitored using the previous JavaMOP system, but it can be monitored with the new one.

15.2.6 Success ratio

Consider now parametric traces with events $\text{success}\langle a \rangle$ and $\text{fail}\langle a \rangle$, saying whether a certain action a was successful or not. For a given action, a meaningful property can classify its (non-parametric) traces into an infinite number of categories, each representing a success ratio of the given action, which is a (rational) number s/t between 0 and 1, where s is the number of success events in the trace and t is the total number of events in the trace. Then the corresponding parametric property over such parametric traces gives a success ratio for each action. We can specify such a property in JavaMOP and RV by making use of monitor variables and event actions [198]. Indeed, one can add two monitor variables, s and t , and then increment t in each event action and increment s only in the event action of the `success` event. The underlying parametric monitoring algorithm keeps separation between the various s and t monitor variables, one such pair for each distinct action a , guaranteeing that the correct ones will be accessed.

15.3 Mathematical Background: Partial Functions, Least Upper Bounds (lubs) and lub Closures

In this section we first discuss some basic notions of partial functions and least upper bounds of them, then we introduce least upper bounds of sets of partial functions and least upper bound closures of sets of partial functions. This section is rather mathematical. We need these mathematical notions because it turns out that parameter instances are partial maps from the domain of parameters to the domain of parameter values. As shown later, whenever a new parametric event is observed, it needs to be dispatched to the interested parts (trace slices or monitors), and those parts updated accordingly: these informal operations can be rigorously formalized as existence of least upper bounds and least upper bound closures over parameter instances, i.e., partial functions.

We recommend the reader who is only interested in our algorithms but not in the details of our subsequent proofs, to read the first two definitions and then jump to Section 15.4, returning to this section for more mathematical background only when needed.

15.3.1 Partial Functions

A lot of this section should go to Chapter 2

This section discusses partial functions and (least) upper bounds over sets of partial functions. The notions and the results discussed in this section are broadly known, and many of their properties are folklore. They can be found in one shape or another in virtually any book on denotational semantics or domain theory. Since we need only a small subset of notions and results on partial functions and (least) upper bounds in this paper, and since we need to fix a uniform notation anyway, we prefer to define and prove everything we need and hereby also make our paper self-contained.

We think of partial functions as “information carriers”: if a partial function θ is defined on an element x of its domain, then “ θ carries the information $\theta(x)$ about $x \in X$ ”. Some partial functions can carry more information than others; two or more partial functions can, together, carry compatible information, but can also carry incompatible information (when two or more of them disagree on the information they carry for a particular $x \in X$).

Definition 64 *We let $[X \rightarrow V]$ and $[X \multimap V]$ denote the sets of **total** and of **partial functions** from X to V , respectively. The domain of $\theta \in [X \multimap V]$ is the set $\text{Dom}(\theta) = \{x \in X \mid \theta(x) \text{ is defined}\}$. Let $\perp \in [X \multimap V]$ be the map undefined everywhere, that is, $\text{Dom}(\perp) = \emptyset$. If $\theta, \theta' \in [X \multimap V]$ then:*

1. θ and θ' are **compatible** if and only if $\theta(x) = \theta'(x)$ for any $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$;
2. θ is **less informative than** θ' , written $\theta \sqsubseteq \theta'$, if for any $x \in X$, $\theta(x)$ defined implies $\theta'(x)$ also defined and $\theta'(x) = \theta(x)$;
3. θ is **strictly less informative than** θ' , written $\theta \sqsubset \theta'$, when $\theta \sqsubseteq \theta'$ and $\theta \neq \theta'$.

The relation of compatibility is reflexive and symmetric, but not transitive. When $\theta, \theta' \in [X \multimap V]$ are compatible, we let $\theta \sqcup \theta' \in [X \multimap V]$ denote the partial function whose domain is $\text{Dom}(\theta) \cup \text{Dom}(\theta')$ and which is defined as θ or θ' in each element in its domain. The partial function $\theta \sqcup \theta'$ is called the least upper bound of θ and θ' . We define least upper bounds more generally below. Also, note that $\theta \sqsubseteq \theta'$ and, respectively, $\theta \sqsubset \theta'$, iff θ, θ' compatible and $\text{Dom}(\theta) \subseteq \text{Dom}(\theta')$ and, respectively, $\text{Dom}(\theta) \subset \text{Dom}(\theta')$.

Definition 65 Given $\Theta \subseteq [X \rightarrow V]$ and $\theta' \in [X \rightarrow V]$,

1. θ' is an **upper bound** of Θ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$; Θ **has upper bounds** iff there is a θ' which is an upper bound of Θ ;
2. θ' is the **least upper bound (lub)** of Θ iff θ' is an upper bound of Θ and $\theta' \sqsubseteq \theta''$ for any other upper bound θ'' of Θ ;
3. θ' is the **maximum (max)** of Θ iff $\theta' \in \Theta$ and θ' is a lub of Θ .

A set of partial functions has an upper bound iff the partial functions in the set are pairwise compatible, that is, no two of them disagree on the value of any particular element in their domain. Least upper bounds and maximums may not always exist for any $\Theta \subseteq [X \rightarrow V]$; if a lub or a maximum for Θ exists, then it is, of course, unique, because \sqsubseteq is a partial order, so antisymmetric.

It is known that $([X \rightarrow V], \sqsubseteq, \perp)$ is a complete (i.e., any \sqsubseteq -chain has a least upper bound) partial order with bottom (i.e., \perp).

Definition 66 Given $\Theta \subseteq [X \rightarrow V]$, let $\sqcup \Theta$ and $\max \Theta$ be the lub and the max of Θ , respectively, when they exist. When Θ is finite, one may write $\theta_1 \sqcup \theta_2 \sqcup \dots \sqcup \theta_n$ instead of $\sqcup \{\theta_1, \theta_2, \dots, \theta_n\}$.

If Θ has a maximum, then it also has a lub and $\sqcup \Theta = \max \Theta$. Here are several common properties that we use frequently in Sections 15.3.2 and 15.3.3 (these sections will present less known results with specific to our particular approach to parametric slicing and monitoring):

Proposition 22 The following hold ($\theta, \theta_1, \theta_2, \theta_3 \in [X \rightarrow V]$): $\perp \sqcup \theta$ exists and $\perp \sqcup \theta = \theta$; $\theta_1 \sqcup \theta_2$ exists iff $\theta_2 \sqcup \theta_1$ exists, and, if they exist then $\theta_1 \sqcup \theta_2 = \theta_2 \sqcup \theta_1$; $\theta_1 \sqcup (\theta_2 \sqcup \theta_3)$ exists iff $(\theta_1 \sqcup \theta_2) \sqcup \theta_3$ exists, and if they exist then $\theta_1 \sqcup (\theta_2 \sqcup \theta_3) = (\theta_1 \sqcup \theta_2) \sqcup \theta_3$.

Proposition 23 Let $\Theta \subseteq [X \rightarrow V]$. Then

1. Θ has an upper bound iff for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ are defined then $\theta_1(x) = \theta_2(x)$;
2. If Θ has an upper bound then $\sqcup \Theta$ exists and, for any $x \in X$,

$$(\sqcup \Theta)(x) = \begin{cases} \text{undefined} & \text{if } \theta(x) \text{ is undefined for any } \theta \in \Theta \\ \theta(x) & \text{if there is a } \theta \in \Theta \text{ with } \theta(x) \text{ defined.} \end{cases}$$

Proof: Since Θ has an upper bound $\theta' \in [X \rightarrow V]$ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$, if $\theta_1, \theta_2 \in \Theta$ and $x \in X$ are such that $\theta_1(x)$ and $\theta_2(x)$ are defined then $\theta'(x)$ is also defined and $\theta_1(x) = \theta_2(x) = \theta'(x)$. Suppose now that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ are defined then $\theta_1(x) = \theta_2(x)$. All we need to show in order to prove both results is that we can find a lub for Θ . Let $\theta' \in [X \rightarrow V]$ be defined as follows: for any $x \in X$, let

$$\theta'(x) = \begin{cases} \text{undefined} & \text{if } \theta(x) \text{ is undefined for any } \theta \in \Theta \\ \theta(x) & \text{if there is a } \theta \in \Theta \text{ with } \theta(x) \text{ defined} \end{cases}$$

First, θ' above is indeed well-defined, because we assumed that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ are defined then $\theta_1(x) = \theta_2(x)$. Second, θ' is an upper bound for Θ : indeed, if $\theta \in \Theta$ and $x \in X$ such that $\theta(x)$ is defined, then $\theta'(x)$ is also defined and $\theta'(x) = \theta(x)$, that is, $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$. Finally, θ' is a lub for Θ : if θ'' is another upper bound for Θ and $\theta'(x)$ is defined for some $x \in X$, that is, $\theta(x)$ is defined for some $\theta \in \Theta$ and $\theta'(x) = \theta(x)$, then $\theta''(x)$ is also defined and $\theta'(x) = \theta(x)$ (as $\theta \sqsubseteq \theta''$), so $\theta' \sqsubseteq \theta''$. \square

Proposition 24 *The following hold:*

1. *The empty set of partial functions $\emptyset \subseteq [X \rightarrow V]$ has upper bounds and $\sqcup \emptyset = \perp$;*
2. *The one-element sets have upper bounds and $\sqcup \{\theta\} = \theta$ for any $\theta \in [X \rightarrow V]$;*
3. *The bottom “ \perp ” does not influence the least upper bounds: $\sqcup(\{\perp\} \cup \Theta) = \sqcup \Theta$ for any $\Theta \subseteq [X \rightarrow V]$;*
4. *If $\Theta, \Theta' \subseteq [X \rightarrow V]$ such that $\sqcup \Theta'$ exists and for any $\theta \in \Theta$ there is a $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, then $\sqcup \Theta$ exists and $\sqcup \Theta \sqsubseteq \sqcup \Theta'$; for example, if $\sqcup \Theta'$ exists and $\Theta \subseteq \Theta'$ then $\sqcup \Theta$ exists and $\sqcup \Theta \sqsubseteq \sqcup \Theta'$;*
5. *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial functions with $\Theta_i \subseteq [X \rightarrow V]$. Then $\sqcup \cup \{\Theta_i \mid i \in I\}$ exists iff $\sqcup \{\sqcup \Theta_i \mid i \in I\}$ exists, and, if both exist,*

$$\sqcup \cup \{\Theta_i \mid i \in I\} = \sqcup \{\sqcup \Theta_i \mid i \in I\}.$$

Proof: 1., 2. and 3. are straightforward. For 4., since for each $\theta \in \Theta$ there is some $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, and since $\theta' \sqsubseteq \sqcup \Theta'$ for any $\theta' \in \Theta'$, it

follows that $\theta \sqsubseteq \sqcup \Theta'$ for any $\theta \in \Theta$, that is, that $\sqcup \Theta'$ is an upper bound for Θ . Therefore, by Proposition 23 it follows that $\sqcup \Theta$ exists and $\sqcup \Theta \sqsubseteq \sqcup \Theta'$ (the latter because $\sqcup \Theta$ is the *least* upper bound of Θ). We prove 5. by double implication, each implication stating that if one of the lub's exist then the other one also exists and one of the inclusions holds; that indeed implies that one of the lub's exists if and only if the other one exists and, if both exist, then they are equal. Suppose first that $\sqcup \cup \{\Theta_i \mid i \in I\}$ exists, that is, that $\cup \{\Theta_i \mid i \in I\}$ has an upper bound, say u . Since $\Theta_i \subseteq \cup \{\Theta_i \mid i \in I\}$ for each $i \in I$, it follows first that each Θ_i also has u as an upper bound, so all $\sqcup \Theta_i$ for all $i \in I$ exist, and second by 4. above that $\sqcup \Theta_i \sqsubseteq \sqcup \cup \{\Theta_i \mid i \in I\}$ for each $i \in I$. Item 4. above then further implies that $\sqcup \{\sqcup \Theta_i \mid i \in I\}$ exists and $\sqcup \{\sqcup \Theta_i \mid i \in I\} \sqsubseteq \sqcup \{\sqcup \cup \{\Theta_i \mid i \in I\}\} = \sqcup \cup \{\Theta_i \mid i \in I\}$ (the last equality follows by 2. above). Conversely, suppose now that $\sqcup \{\sqcup \Theta_i \mid i \in I\}$ exists. Since for each $\theta \in \cup \{\Theta_i \mid i \in I\}$ there is some $i \in I$ such that $\theta \sqsubseteq \sqcup \Theta_i$ (an $i \in I$ such that $\theta \in \Theta_i$), item 4. above implies that $\sqcup \cup \{\Theta_i \mid i \in I\}$ also exists and $\sqcup \cup \{\Theta_i \mid i \in I\} \sqsubseteq \sqcup \{\sqcup \Theta_i \mid i \in I\}$. \square

15.3.2 Least Upper Bounds of Families of Sets of Partial Maps

Motivated by requirements and optimizations of our trace slicing and monitoring algorithms in Sections 15.6 and 15.8, in this section and the next we define several less known notions and results. We are actually not aware of other places where these notions are defined, so they could be novel and specific to our approach to parametric trace slicing and monitoring.

We first extend the notion of least upper bound (lub) from one associating a partial function to a set of partial functions to one associating a set of partial functions to a family (or set) of sets of partial functions:

Definition 67 *If $\{\Theta_i\}_{i \in I}$ is a family of sets in $[X \rightarrow V]$, then we let the **least upper bound** (also **lub**) of $\{\Theta_i\}_{i \in I}$ be defined as:*

$$\sqcup \{\Theta_i \mid i \in I\} \stackrel{\text{def}}{=} \{\sqcup \{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i \text{ for each } i \in I \text{ such that } \sqcup \{\theta_i \mid i \in I\} \text{ exists}\}.$$

As before, we use the infix notation when I is finite, e.g., we may write $\Theta_1 \sqcup \Theta_2 \sqcup \dots \sqcup \Theta_n$ instead of $\sqcup \{\Theta_i \mid i \in \{1, 2, \dots, n\}\}$.

Therefore, $\sqcup \{\Theta_i \mid i \in I\}$ is the set containing all the lub's corresponding to sets formed by picking for each $i \in I$ precisely one element from Θ_i .

Unlike for sets of partial functions, the lub's of families of sets of partial functions always exist; $\sqcup\{\Theta_i \mid i \in I\}$ is the empty set when no collection of $\theta_i \in \Theta_i$ can be found (one $\theta_i \in \Theta_i$ for each $i \in I$) such that $\{\theta_i \mid i \in I\}$ has an upper bound.

There is an admitted slight notational ambiguity between the two least upper bound notations introduced so far. We prefer to purposely allow this ambiguity instead of inventing a new notation for the lub's of families of sets, hoping that the reader is able to quickly disambiguate the two by checking the types of the objects involved in the lub: if partial functions then the first lub is meant, if sets of partial functions then the second. Note that such notational ambiguities are actually common practice elsewhere; e.g., in a monoid $(M, *_- : M \times M \rightarrow M, 1)$ with binary operation $*$ and unit 1, the $*$ is commonly extended to sets of elements M_1, M_2 in M as expected: $M_1 * M_2 = \{m_1 * m_2 \mid m_1 \in M_1, m_2 \in M_2\}$.

Proposition 25 *The following facts hold, where $\Theta, \Theta_1, \Theta_2, \Theta_3 \subseteq [X \rightarrow V]$:*

1. $\sqcup\emptyset = \{\perp\}$, where, in this case, the empty set $\emptyset \subseteq \mathcal{P}([X \rightarrow V])$ is meant;
2. $\sqcup\{\Theta\} = \Theta$; in particular $\sqcup\{\emptyset\} = \emptyset$ when the empty set $\emptyset \subseteq [X \rightarrow V]$ is meant;
3. $\sqcup\{\{\theta\} \mid \theta \in \Theta\} = \begin{cases} \{\sqcup\Theta\} & \text{if } \Theta \text{ has a lub, and} \\ \emptyset & \text{if } \Theta \text{ does not have a lub;} \end{cases}$
4. $\emptyset \sqcup \Theta = \emptyset$, where the empty set $\emptyset \subseteq [X \rightarrow V]$ is meant;
5. $\{\perp\} \sqcup \Theta = \Theta$;
6. If $\Theta_1 \subseteq \Theta_2$ then $\Theta_1 \sqcup \Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$; in particular, if $\perp \in \Theta_2$ then $\Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$;
7. $(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 = (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$.

Proof: Recall that the least upper bound $\sqcup\{\Theta_i \mid i \in I\}$ of sets of sets of partial functions is built by collecting all the lubs of sets $\{\theta_i \mid i \in I\}$ containing one element θ_i from each of the sets Θ_i . When $|I| = 0$, that is when I is empty, there is precisely one set $\{\theta_i \mid i \in I\}$, the empty set of partial functions. Then 1. follows by 1. in Proposition 24. When $|I| = 1$, that is when $\{\Theta_i \mid i \in I\} = \{\Theta\}$ for some $\Theta \subseteq [X \rightarrow V]$ like in 2., then the sets $\{\theta_i \mid i \in I\}$ are precisely the singleton sets corresponding to the elements of Θ , so 2. follows by 2. in Proposition 24. 3. holds because there

is only one way to pick an element from each singleton set $\{\theta\}$, namely to pick the θ itself; this also shows how the notion of a lub of a family of sets generalizes the conventional notion of lub. When $|I| \geq 2$ and at least one of the involved sets of partial functions is empty, like in 4., then there is no set $\{\theta_i \mid i \in I\}$, so the least upper bound of the set of sets is empty (regarded, again, as the empty set of sets of partial functions). 5. follows by 1. in Proposition 22. The first part of 6. is immediate and the second part follows from the first using 5.. Finally, 7. follows by double implication: $(\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3) \subseteq (\Theta_1 \cup \Theta_2) \sqcup \Theta_3$ follows by 6. because Θ_1 and Θ_2 are included in $\Theta_1 \cup \Theta_2$, and $(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 \subseteq (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$ because for any $\theta_1 \in \Theta_1 \cup \Theta_2$, say $\theta_1 \in \Theta_1$, and any $\theta_3 \in \Theta_3$, if $\theta_1 \sqcup \theta_3$ exists then it also belongs to $\Theta_1 \sqcup \Theta_3$. \square

Proposition 26 *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial maps in $[X \rightarrow V]$ and let $\mathcal{I} = \{I_j\}_{j \in J}$ be a partition of I : $I = \cup\{I_j \mid j \in J\}$ and $I_{j_1} \cap I_{j_2} = \emptyset$ for any different $j_1, j_2 \in J$. Then $\sqcup\{\Theta_i \mid i \in I\} = \sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$.*

Proof: For each $j \in J$, let Q_j denote the set $\sqcup\{\Theta_{i_j} \mid i_j \in I_j\}$. Definition 67 then implies the following: $Q_j \stackrel{\text{def}}{=} \{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid \theta_{i_j} \in \Theta_{i_j} \text{ for each } i_j \in I_j, \text{ such that } \sqcup\{\theta_{i_j} \mid i_j \in I_j\} \text{ exists}\}$. Definition 67 also implies the following: $\sqcup\{Q_j \mid j \in J\} \stackrel{\text{def}}{=} \{\sqcup\{q_j \mid j \in J\} \mid q_j \in Q_j \text{ for each } j \in J, \text{ such that } \sqcup\{q_j \mid j \in J\} \text{ exists}\}$. Putting the two equalities above together, we get that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals the following:

$$\left\{ \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\} \mid \begin{array}{l} \theta_{i_j} \in \Theta_{i_j} \text{ for each } j \in J \text{ and } i_j \in I_j, \text{ such that} \\ \sqcup\{\theta_{i_j} \mid i_j \in I_j\} \text{ exists for each } j \in J \text{ and} \\ \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\} \text{ exists} \end{array} \right\}.$$

Since $\{I_j\}_{j \in J}$ is a partition of I , the indexes i_j generated by “for each $j \in J$ and $i_j \in I_j$ ” cover precisely all the indexes $i \in I$. Moreover, picking partial functions $\theta_{i_j} \in \Theta_{i_j}$ for each $j \in J$ and $i_j \in I_j$ is equivalent to picking partial functions $\theta_i \in \Theta_i$ for each $i \in I$, and, in this case, $\{\theta_i \mid i \in I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$. By 5. in Proposition 24 we then infer that $\sqcup\{\theta_i \mid i \in I\}$ exists if and only if $\sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ exists, and if both exist then $\sqcup\{\theta_i \mid i \in I\} = \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$; if both exist then $\sqcup\{\theta_{i_j} \mid i_j \in I_j\}$ also exists for each $j \in J$ (because $\{\theta_{i_j} \mid i_j \in I_j\} \subseteq \{\theta_i \mid i \in I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$). Therefore, we can conclude that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals $\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i \text{ for each } i \in I, \text{ such that } \sqcup\{\theta_i \mid i \in I\} \text{ exists}\}$, which is nothing but $\sqcup\{\Theta_i \mid i \in I\}$. \square

Corollary 4 *The following hold:*

1. $\{\perp\} \sqcup \Theta = \Theta$ (already proved as 5. in Proposition 25);
2. $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1$;
3. $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3$;

Proof: These follow from Proposition 26 for various index sets I and partitions of it: for 1. take $I = \{1\}$ and its partition $I = \emptyset \cup I$, take $\Theta_1 = \Theta$, and then use 1. in Proposition 25 saying that $\sqcup \emptyset = \{\perp\}$; for 2. take partitions $\{1\} \cup \{2\}$ and $\{2\} \cup \{1\}$ of $I = \{1, 2\}$, getting $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1 = \sqcup \{\Theta_i \mid i \in \{1, 2\}\}$; finally, for 3. take partitions $\{1\} \cup \{2, 3\}$ and $\{1, 2\} \cup \{3\}$ of $I = \{1, 2, 3\}$, getting $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3 = \sqcup \{\Theta_i \mid i \in \{1, 2, 3\}\}$. \square

15.3.3 Least Upper Bound Closures

We next define lub closures of sets of partial maps, a crucial operation for the algorithms discussed next in the paper.

Definition 68 $\Theta \subseteq [X \rightarrow V]$ is **lub closed** if and only if $\sqcup \Theta' \in \Theta$ for any $\Theta' \subseteq \Theta$ admitting upper bounds.

Proposition 27 $\{\perp\}$ and $\{\perp, \theta\}$ are lub closed ($\theta \in [X \rightarrow V]$).

Proof: It follows easily from Definition 68, using the facts that $\sqcup \emptyset = \perp$ (1. in Proposition 24), $\sqcup \{\theta\} = \theta$ (2. in Proposition 24), and $\sqcup \{\perp, \theta\} = \theta$ (3. in Proposition 24 for $\Theta = \{\theta\}$). \square

Proposition 28 If $\Theta \subseteq [X \rightarrow V]$ and $\{\Theta_i\}_{i \in I}$ is a family of sets of partial functions in $[X \rightarrow V]$, then:

1. If Θ is lub closed then $\perp \in \Theta$; in particular, \emptyset is not lub closed;
2. If Θ has upper bounds and is lub closed then it has a maximum;
3. Θ is lub closed iff $\sqcup \{\Theta \mid i \in I\} = \Theta$ for any I ;
4. If Θ is lub closed and $\Theta_i \subseteq \Theta$ for each $i \in I$ then $\sqcup \{\Theta_i \mid i \in I\} \subseteq \Theta$;
5. If Θ_i is lub closed for each $i \in I$ then $\sqcup \{\Theta_i \mid i \in I\}$ is lub closed and

$$\sqcup \{\Theta_i \mid i \in I\} \subseteq \sqcup \{\Theta_i \mid i \in I\};$$

6. If I finite and Θ_i finite for all $i \in I$, then $\sqcup\{\Theta_i \mid i \in I\}$ finite;
7. If Θ_i lub closed for all $i \in I$ then $\sqcup\{\Theta_i \mid i \in I\}$ is lub closed;
8. $\sqcap\{\Theta' \mid \Theta' \subseteq [X \rightarrow V] \text{ with } \Theta \subseteq \Theta' \text{ and } \Theta' \text{ is lub closed}\}$ is the smallest lub closed set including Θ .

Proof: 1. follows taking $\Theta' = \emptyset$ in Definition 68 and using $\sqcup\emptyset = \perp$ (1. in Proposition 24).

2. follows taking $\Theta' = \Theta$ in Definition 68: $\sqcup\Theta \in \Theta$, so $\max \Theta$ exists (and equals $\sqcup\Theta$).

3. Definition 67 implies that $\sqcup\{\Theta \mid i \in I\}$ equals $\{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta \text{ for each } i \in I, \text{ such that } \sqcup\{\theta_i \mid i \in I\} \text{ exists}\}$, which is nothing but $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta \text{ such that } \sqcup\Theta' \text{ exists}\}$; the later can be now shown equal to Θ by double inclusion: $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta \text{ such that } \sqcup\Theta' \text{ exists}\} \subseteq \Theta$ because Θ is lub closed, and $\Theta \subseteq \{\sqcup\Theta' \mid \Theta' \subseteq \Theta \text{ such that } \sqcup\Theta' \text{ exists}\}$ because one can pick $\Theta' = \{\theta\}$ for each $\theta \in \Theta$ and use the fact that $\sqcup\{\theta\} = \theta$ (2. in Proposition 24).

4. Let θ be an arbitrary partial function in $\sqcup\{\Theta_i \mid i \in I\}$, that is, $\theta = \sqcup\{\theta_i \mid i \in I\}$ for some $\theta_i \in \Theta_i$, one for each $i \in I$, such that $\{\theta_i \mid i \in I\}$ has upper bounds. Since Θ is lub closed and $\Theta_i \subseteq \Theta$ for each $i \in I$, it follows that $\theta \in \Theta$. Therefore, $\sqcup\{\Theta_i \mid i \in I\} \subseteq \Theta$.

5. Let Θ' be a set of partial functions included in $\sqcup\{\Theta_i \mid i \in I\}$ which admits an upper bound; moreover, for each $\theta' \in \Theta'$, let us fix a set $\{\theta_i^{\theta'} \mid i \in I\}$ such that $\theta_i^{\theta'} \in \Theta_i$ for each $i \in I$ and $\theta' = \sqcup\{\theta_i^{\theta'} \mid i \in I\}$ (such sets exist because $\theta' \in \Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$). Let $\Theta^{\theta'}$ be the set $\{\theta_i^{\theta'} \mid i \in I\}$ for each $\theta' \in \Theta'$, let Θ'_i be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta'\}$ for each $i \in I$, and let Θ be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta', i \in I^{\theta'}\}$. It is easy to see that $\Theta = \sqcup\{\Theta^{\theta'} \mid \theta' \in \Theta'\} = \sqcup\{\Theta'_i \mid i \in I\}$ and that $\Theta'_i \subseteq \Theta_i$ for each $i \in I$. Since $\sqcup\Theta'$ exists (because Θ' has upper bounds) and $\sqcup\Theta' = \sqcup\{\theta' \mid \theta' \in \Theta'\} = \sqcup\{\sqcup\Theta^{\theta'} \mid \theta' \in \Theta'\}$, by 5. in Proposition 24 it follows that $\sqcup\Theta$ exists and $\sqcup\Theta' = \sqcup\Theta$. Since $\Theta = \sqcup\{\Theta'_i \mid i \in I\}$ and $\sqcup\Theta$ exists, by 5. in Proposition 24 again we get that $\sqcup\{\sqcup\Theta'_i \mid i \in I\}$ exists and is equal to $\sqcup\Theta$, which is equal to $\sqcup\Theta'$. Since $\Theta'_i \subseteq \Theta_i$ and Θ_i is lub closed, we get that $\sqcup\Theta'_i \in \Theta_i$. That means that $\sqcup\{\sqcup\Theta'_i \mid i \in I\} \in \sqcup\{\Theta_i \mid i \in I\}$, that is, that $\sqcup\Theta' \in \sqcup\{\Theta_i \mid i \in I\}$. Since $\Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$ was chosen arbitrarily, we conclude that $\sqcup\{\Theta_i \mid i \in I\}$ is lub closed. To show that $\sqcup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$, let us pick an $i \in I$ and let us partition I as $\{i\} \cup (I \setminus \{i\})$. By Proposition 26, $\sqcup\{\Theta_i \mid i \in I\} = \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I \setminus \{i\}\})$. The proof above also implies that $\sqcup\{\Theta_j \mid j \in I \setminus \{i\}\}$ is lub closed, so by 1.

we get that $\perp \in \sqcup\{\Theta_j \mid j \in I \setminus \{i\}\}$. Finally, 6. in Proposition 25 implies $\Theta_i \sqsubseteq \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I \setminus \{i\}\})$, so $\Theta_i \subseteq \sqcup\{\Theta_i \mid i \in I\}$ for each $i \in I$, that is, $\sqcup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$.

6. Recall from Definition 67 that $\sqcup\{\Theta_i \mid i \in I\}$ contains the existing least upper bounds of sets of partial functions containing precisely one partial function in each Θ_i . If I and each of the Θ_i for each $i \in I$ is finite, then $|\sqcup\{\Theta_i \mid i \in I\}| \leq \prod_{i \in I} |\Theta_i|$, because there at most $\prod_{i \in I} |\Theta_i|$ combinations of partial functions, one in each Θ_i , that admit an upper bound. Therefore, $\sqcup\{\Theta_i \mid i \in I\}$ is also finite.

7. Let $\Theta' \subseteq \cap\{\Theta_i \mid i \in I\}$ be a set of partial functions admitting an upper bound. Then $\Theta' \subseteq \Theta_i$ for each $i \in I$ and, since each Θ_i is lub closed, $\sqcup\Theta' \in \Theta_i$ for each $i \in I$. Therefore, $\sqcup\Theta' \in \cap\{\Theta_i \mid i \in I\}$.

8. Anticipating the definition of and notation for lub closures (Definition 68), we let $\overline{\Theta}$ denote the set $\cap\{\Theta' \mid \Theta' \subseteq [X \rightarrow V] \text{ with } \Theta \subseteq \Theta' \text{ and } \Theta' \text{ is lub closed}\}$. It is clear that $\Theta \subseteq \overline{\Theta}$ and, by 7., that $\overline{\Theta}$ is lub closed. It is also the *smallest* lub closed set including Θ , because all such sets Θ' are among those whose intersection defines $\overline{\Theta}$. \square

Definition 69 Given $\theta' \in [X \rightarrow V]$ and $\Theta \subseteq [X \rightarrow V]$, let

$$(\theta')_{\Theta} \stackrel{\text{def}}{=} \{\theta \mid \theta \in \Theta \text{ and } \theta \sqsubseteq \theta'\}$$

be the set of partial functions in Θ that are less informative than θ' .

Proposition 29 If $\theta, \theta', \theta'', \theta_1, \theta_2 \in [X \rightarrow V]$ and if $\Theta \subseteq [X \rightarrow V]$ is lub closed, then:

1. $(\theta')_{\Theta}$ is lub closed and $\max(\theta')_{\Theta}$ exists;
2. If $\theta' \in \{\theta\} \sqcup \Theta$ then $\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta' = \theta \sqcup \theta''\}$ has maximum and that equals $\max(\theta')_{\Theta}$;
3. If $\theta_1, \theta_2 \in \{\theta\} \sqcup \Theta$ such that $\theta_1 = \max(\theta_2)_{\Theta}$, then $\theta_1 = \theta_2$.

Proof: 1. First, note that θ' is an upper bound for $(\theta')_{\Theta}$ as well as for any subset Θ' of it, so any $\Theta' \subseteq (\theta')_{\Theta}$ has upper bounds, so by 2. in Proposition 23, $\sqcup\Theta'$ exists for any $\Theta' \subseteq (\theta')_{\Theta}$. Moreover, if $\Theta' \subseteq (\theta')_{\Theta}$ then $\sqcup\Theta' \sqsubseteq \theta'$, and since Θ is lub closed it follows that $\sqcup\Theta' \in \Theta$, so $\sqcup\Theta' \in (\theta')_{\Theta}$. Therefore, $(\theta')_{\Theta}$ is lub closed. 2. in Proposition 28 now implies that $(\theta')_{\Theta}$ has maximum; to be concrete, $\max(\theta')_{\Theta}$ is nothing but $\sqcup(\theta')_{\Theta}$, which belongs to $(\theta')_{\Theta}$ (because one can pick $\Theta' = (\theta')_{\Theta}$ above).

2. Let Q be the set of partial functions $\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta' = \theta \sqcup \theta''\}$. Note that Q is non-empty (because $\theta' \in \{\theta\} \sqcup \Theta$, so there is some $\theta'' \in \Theta$ such as $\theta' = \theta \sqcup \theta''$) and has upper-bounds (because θ' is an upper bound for it), but that it is not necessarily lub closed (because, unless $\theta' = \theta$, Q does not contain \perp , contradicting 1. in Proposition 28). Hence Q has a lub (by 2. in Proposition 23), say q , and $q = \sqcup Q \sqsubseteq \theta'$; since $\theta \sqsubseteq \theta'$, it follows that $\theta \sqcup q \sqsubseteq \theta'$. On the other hand $\theta' \sqsubseteq \theta \sqcup q$ by 4. in Proposition 24, because there is some $\theta'' \in Q$ such that $\theta' = \theta \sqcup \theta''$ and $\theta'' \sqsubseteq q$. Therefore, $\theta' = \theta \sqcup q$. Since Θ is lub closed, it follows that $q \in \Theta$. Therefore, $q \in Q$, so q is the maximum element of Q . Let us next show that $q = \max(\theta']_{\Theta}$. The relation $q \sqsubseteq \max(\theta']_{\Theta}$ is immediate because $q \in (\theta']_{\Theta}$ (we proved above that $q \in \Theta$ and $q \sqsubseteq \theta'$). For $\max(\theta']_{\Theta} \sqsubseteq q$ it suffices to show that $\max(\theta']_{\Theta} \in Q$, that is, that $\theta \sqcup \max(\theta']_{\Theta} = \theta'$: $\theta \sqcup \max(\theta']_{\Theta} \sqsubseteq \theta'$ follows because $\theta \sqsubseteq \theta'$ and $\max(\theta']_{\Theta} \sqsubseteq \theta'$, while $\theta' \sqsubseteq \theta \sqcup \max(\theta']_{\Theta}$ follows because there is some $\theta'' \in \Theta$ such that $\theta' = \theta \sqcup \theta''$ and, since $\theta'' \sqsubseteq \max(\theta']_{\Theta}$, $\theta \sqcup \theta'' \sqsubseteq \theta \sqcup \max(\theta']_{\Theta}$ (by 4. in Proposition 24).

3. admits a direct proof simpler than that of 2.; however, since 2. is needed anyway, we prefer to use 2. Note that $\theta \sqsubseteq \theta_1 \sqsubseteq \theta_2$. By 2., $\theta_1 = \max\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta_2 = \theta \sqcup \theta''\}$, which implies $\theta_2 = \theta \sqcup \theta_1 = \theta_1$. \square

Definition 70 Given $\Theta \subseteq [X \rightarrow V]$, we let $\overline{\Theta}$, the *least upper bound (lub) closure* of Θ , be defined as follows:

$$\overline{\Theta} \stackrel{\text{def}}{=} \cap\{\Theta' \mid \Theta' \subseteq [X \rightarrow V] \text{ with } \Theta \subseteq \Theta' \text{ and } \Theta' \text{ is lub closed}\}.$$

Proposition 30 The next hold, where $\Theta \subseteq [X \rightarrow V]$ and $\theta \in [X \rightarrow V]$:

1. $\overline{\Theta}$ is the smallest lub closed set including Θ ;
2. $\overline{\emptyset} = \overline{\{\perp\}} = \{\perp\}$;
3. $\overline{\{\theta\}} = \{\perp, \theta\}$.

Proof: 1. follows by 7. in Proposition 28. For 2. and 3., first note that $\{\perp\}$ and $\{\perp, \theta\}$ are lub closed by Proposition 27; second, note that they are indeed the smallest lub closed sets including \perp and resp. θ , as any lub closed set must include \perp (1. in Proposition 28). \square

Proposition 31 The lub closure map $\bar{\cdot} : 2^{[X \rightarrow V]} \rightarrow 2^{[X \rightarrow V]}$ is a closure operator, that is, for any $\Theta, \Theta_1, \Theta_2 \subseteq [X \rightarrow V]$,

1. (*extensivity*) $\Theta \subseteq \overline{\Theta}$;
2. (*monotonicity*) If $\Theta_1 \subseteq \Theta_2$ then $\overline{\Theta_1} \subseteq \overline{\Theta_2}$;
3. (*idempotency*) $\overline{\overline{\Theta}} = \overline{\Theta}$.

Proof: Extensivity and idempotency follow immediately from the definitions of $\overline{\Theta}$ and $\overline{\overline{\Theta}}$ (which are lub closed by 1. in Proposition 30). For monotonicity, one should note that $\overline{\Theta_2}$ satisfies the properties of $\overline{\Theta_1}$ (i.e., $\Theta_1 \subseteq \overline{\Theta_2}$ and Θ_2 is lub closed); since $\overline{\Theta_1}$ is the smallest with those properties, it follows that $\overline{\Theta_1} \subseteq \overline{\Theta_2}$. \square

Proposition 32 $\overline{\cup\{\Theta_i \mid i \in I\}} = \sqcup\{\overline{\Theta_i} \mid i \in I\}$ for any family $\{\Theta_i\}_{i \in I}$ of partial functions in $[X \rightarrow V]$.

Proof: Since $\overline{\Theta_i}$ is lub closed for any $i \in I$, 5. in Proposition 28 implies that $\sqcup\{\overline{\Theta_i} \mid i \in I\}$ is lub closed and $\cup\{\overline{\Theta_i} \mid i \in I\} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$. Since 1. in Proposition 31 implies $\Theta_i \subseteq \overline{\Theta_i}$ for each $i \in I$ and since $\overline{\cup\{\Theta_i \mid i \in I\}}$ is the smallest lub closed set including $\cup\{\Theta_i \mid i \in I\}$ (1. in Proposition 30), the inclusion $\overline{\cup\{\Theta_i \mid i \in I\}} \subseteq \sqcup\{\overline{\Theta_i} \mid i \in I\}$ holds. Conversely, 2. in Proposition 31 implies that $\overline{\Theta_i} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ for any $i \in I$, so $\sqcup\{\overline{\Theta_i} \mid i \in I\} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ holds by 4. in Proposition 28. \square

Corollary 5 For any $\theta \in [X \rightarrow V]$ and $\Theta \subseteq [X \rightarrow V]$, equality $\overline{\{\theta\} \cup \Theta} = \{\perp, \theta\} \sqcup \overline{\Theta}$ holds.

Proof: $\overline{\{\theta\} \cup \Theta} = \overline{\{\theta\}} \sqcup \overline{\Theta}$ by Proposition 32, and $\overline{\{\theta\}} = \{\perp, \theta\}$ by 3. in Proposition 30. \square

Corollary 6 If $\Theta \subseteq [X \rightarrow V]$ is finite then $\overline{\Theta}$ is also finite.

Proof: Suppose that $\Theta = \{\theta_1, \theta_2, \dots, \theta_n\}$ for some $n \geq 0$. Iteratively applying Corollary 5, $\overline{\Theta} = \{\perp, \theta_1\} \sqcup \{\perp, \theta_2\} \sqcup \dots \sqcup \{\perp, \theta_n\}$; in obtaining that, we used 2. in Proposition 30 and 1. in Corollary 4. The result follows now by 6. in Proposition 28. \square

15.4 Parametric Traces and Properties

Here we introduce the notions of event, trace and property, first non-parametric and then parametric. Traces are sequences of events. Parametric

events can carry data-values as instances of parameters. Parametric traces are traces over parametric events. Properties are trace classifiers, that is, mappings partitioning the space of traces into categories (violating traces, validating traces, don't know traces, etc.). Parametric properties are parametric trace classifiers and provide, for each parameter instance, the category to which the trace slice corresponding to that parameter instance belongs. Trace slicing is defined as a reduct operation that forgets all the events that are unrelated to the given parameter instance.

15.4.1 The Non-Parametric Case

We next introduce non-parametric events, traces and properties, which will serve as an infrastructure for their parametric variants in Section 15.4.2.

Definition 71 *Let \mathcal{E} be a set of (non-parametric) events, called **base events** or simply **events**. An \mathcal{E} -**trace**, or simply a (non-parametric) **trace** when \mathcal{E} is understood or not important, is any finite sequence of events in \mathcal{E} , that is, an element in \mathcal{E}^* . If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

Our parametric trace slicing and monitoring techniques in Sections 15.6 and 15.8 can be easily adapted to also work with infinite traces. Since infinite versus finite traces is not an important aspect of the work reported here, we keep the presentation free of unnecessary technical complications and consider only finite traces.

Example 21 *(part 1 of simple running example) Like in Section 15.2.1, consider a certain resource (e.g., a synchronization object) that can be acquired and released during the lifetime of a given procedure (between its begin and end). Then the set of events \mathcal{E} is $\{\text{acquire}, \text{release}, \text{begin}, \text{end}\}$ and execution traces corresponding to this resource are sequences *begin acquire acquire release end begin end*, *begin acquire acquire*, *begin acquire release acquire end*, etc. For the time being, there are no “good” or “bad” execution traces.*

There is a plethora of formalisms to specify trace requirements. Many of these result in specifying at least two types of traces: those *validating* the specification (i.e, correct traces), and those *violating* the specification (i.e., incorrect traces).

Example 22 (part 2) Consider a regular expression specification,

$$(\text{begin}(\epsilon \mid (\text{acquire}(\text{acquire} \mid \text{release})^* \text{release}))\text{end})^*$$

stating that the procedure can (non-recursively) take place multiple times and, if the resource is acquired during the procedure then it is released by the end of the procedure. Assume that the resource can be acquired and released multiple times, with the effect of acquiring and respectively releasing it precisely once. The validating (or matching) traces for this property are those satisfying the pattern, e.g., *begin acquire acquire release end begin end*. At first sight, one may say that all the other traces are violating (or failing) traces, because they are not in the language of the regular expression. However, there are two interesting types of such “failing” traces: ones which may still lead to a matching trace provided the right events will be received in the future, e.g., *begin acquire acquire*, and ones which have no chance of becoming a matching trace, e.g., *begin acquire release acquire end*.

some of the discussion above unnecessary now, it should just refer to existing material on bad/good prefixes

In general, traces are not enforced to correspond to terminated programs (this is particularly useful in monitoring); if one wants to enforce traces to correspond to terminated programs, then one can have the system generate a special end-of-trace event and have the property specification require that event at the end of each trace.

Therefore, a trace property may partition the space of traces into more than two categories. For some specification formalisms, for example ones based on fuzzy logics or multiple truth values, the set of traces may be split into more than three categories, even into a continuous space of categories. Section 15.2.6 showed an example of a property where the space of trace categories was the set of rational numbers between 0 and 1.

Definition 72 An \mathcal{E} -property P , or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \rightarrow \mathcal{C}$ partitioning the set of traces into (verdict) categories \mathcal{C} .

the view above should also be discussed and incorporated earlier in the book

It is common, though not enforced, that the set of property verdict categories \mathcal{C} in Definition 101 includes `validating` (or similar), `violating` (or similar), and `don't know` (or `?`) categories. In general, \mathcal{C} can be any set, finite or infinite.

We believe that the definition of non-parametric trace property above is general enough that it can easily accommodate any particular specification formalism, such as ones based on linear temporal logics, regular expressions, context-free grammars, etc. All one needs to do in order to instantiate the general results in this paper for a particular specification formalism is to decide upon the desired categories in which traces are intended to be classified, and then define the property associated to a specification accordingly.

For example, if the specification formalism of choice is that of regular expressions over \mathcal{E} and one is interested in classifying traces in three categories as in our example above, then one can pick \mathcal{C} to be the set `{match, fail, don't know}` and, for a given regular expression E , define its associated property $P_E : \mathcal{E}^* \rightarrow \mathcal{C}$ as follows: $P_E(w) = \text{match}$ iff w is in the language of E , $P_E(w) = \text{fail}$ iff there is no $w' \in \mathcal{E}^*$ such that ww' is in the language of E , and $P_E(w) = \text{don't know}$ otherwise; this is the monitoring semantics of regular expressions in the JavaMOP and RV systems [198, 62, 197].

Other semantic choices are possible even for the simple case of regular expressions; for example, one may choose \mathcal{C} to be the set `{match, don't care}` and define $P_E(w) = \text{match}$ iff w is in the language of E , and $P_E(w) = \text{don't care}$ otherwise; this is the semantics of regular expressions in Tracematches [9], where, depending upon how one writes the regular expression, matching can mean either a violation or a validation of the desired property.

Similarly, one can have various verdict categories for linear temporal logic (LTL). For example, one can report `violation` when a bad prefix is encountered, can report `validation` when a good prefix is encountered, and can report `inconclusive` when neither of the above; this is the current monitoring semantics of monitoring LTL in JavaMOP and RV [198, 197]. Semantical and algorithmic aspects regarding LTL monitoring can be found, e.g., in [177, 223, 257, 34].

In some applications, one may not be interested in certain categories of traces, such as in those classified as `don't know`, `don't care`, or `inconclusive`; if that is the case, then those applications can simply ignore these, like Tracematches, JavaMOP and RV do. It may be worth making it explicit that in this paper we do not attempt to propose or promote any particular formalism

for specifying properties about execution traces. Instead, our approach is to define properties as generally as possible to capture the various specification formalisms that we are aware of as special cases, and then to develop our subsequent techniques to work with such general properties. We believe that our definition of property in Definition 101 is general enough to allow us to claim that our results are specification-formalism-independent.

An additional benefit of defining properties so generally, as mappings from traces to categories, is that parametric properties, in spite of their much more general flavor, are also properties (but, obviously, over different traces and over different categories).

15.4.2 The Parametric Case

Events often carry concrete data instantiating parameters.

Example 23 (part 3) *In our running example, events **acquire** and **release** are parametric in the resource being acquired or released; if r is the name of the generic resource parameter and r_1 and r_2 are two concrete resources, then parametric **acquire**/**release** events have the form $\text{acquire}\langle r \mapsto r_1 \rangle$, $\text{release}\langle r \mapsto r_2 \rangle$, etc. Not all events need carry instances for all parameters; e.g., the **begin**/**end** parametric events have the form $\text{begin}\langle \perp \rangle$ and $\text{end}\langle \perp \rangle$, where \perp , the partial map undefined everywhere, instantiates no parameter.*

Recall from Definition 64 that $[A \rightarrow B]$ and $[A \multimap B]$ denote the sets of total and, respectively, partial functions from A to B .

Definition 73 (*Parametric events and traces*). *Let X be a set of **parameters** and let V be a set of corresponding **parameter values**. If \mathcal{E} is a set of base events like in Definition 100, then let $\mathcal{E}\langle X \rangle$ denote the set of corresponding **parametric events** $e\langle \theta \rangle$, where e is a base event in \mathcal{E} and θ is a partial function in $[X \multimap V]$. A **parametric trace** is a trace with events in $\mathcal{E}\langle X \rangle$, that is, a word in $\mathcal{E}\langle X \rangle^*$.*

Therefore, a parametric event is an event carrying values for zero, one, several or even all the parameters, and a parametric trace is a finite sequence of parametric events. In practice, the number of values carried by an event is finite; however, we do not need to enforce this restriction in our theoretical developments. Also, in practice the parameters may be typed, in which case the set of their corresponding values is given by their type. To simplify writing, we occasionally assume the set of parameter values V implicit.

Example 24 (part 4) *A parametric trace for our running example can be the following:*

$\text{begin}\langle\perp\rangle \text{acquire}\langle\theta_1\rangle \text{acquire}\langle\theta_2\rangle \text{acquire}\langle\theta_1\rangle \text{release}\langle\theta_1\rangle \text{end}\langle\perp\rangle \text{begin}\langle\perp\rangle \text{acquire}\langle\theta_2\rangle \text{release}\langle\theta_2\rangle \text{end}\langle\perp\rangle$, where θ_1 maps r to r_1 and θ_2 maps r to r_2 . To simplify writing, we take the freedom to only list the parameter instance values when writing parameter instances, that is, $\langle r_1 \rangle$ instead of $\langle r \mapsto r_1 \rangle$, or \nmid_{r_2} instead of $\nmid_{r \mapsto r_2}$, etc. This notation is formally introduced in the next section, as Notation 1. With this notation, the above trace becomes

$\text{begin}\langle\rangle \text{acquire}\langle r_1 \rangle \text{acquire}\langle r_2 \rangle \text{acquire}\langle r_1 \rangle \text{release}\langle r_1 \rangle \text{end}\langle\rangle \text{begin}\langle\rangle \text{acquire}\langle r_2 \rangle \text{release}\langle r_2 \rangle \text{end}\langle\rangle$.

This trace involves two resources, r_1 and r_2 , and it really consists of two trace slices, one for each resource, merged together. The begin and end events belong to both trace slices. The slice corresponding to θ_1 is

$\text{begin acquire acquire release end begin end}$

while the one for θ_2 is

$\text{begin acquire end begin acquire release end}$.

Recall from Definition 64 that two partial maps of same source and target are compatible when they do not disagree on any of the elements on which they are both defined, and that one is less informative than another, written $\theta \sqsubseteq \theta'$, when they are compatible and the domain of the former is included in the domain of the latter. With the notation in the example above we have, for our running example, that $\langle\rangle$ is compatible with $\langle r_1 \rangle$ and with $\langle r_2 \rangle$, but $\langle r_1 \rangle$ and $\langle r_2 \rangle$ are not compatible; moreover, $\langle\rangle \sqsubseteq \langle r_1 \rangle$ and $\langle\rangle \sqsubseteq \langle r_2 \rangle$.

Definition 74 (Trace slicing) *Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and partial function θ in $[X \rightarrow V]$, we let the θ -trace slice $\tau \upharpoonright_\theta \in \mathcal{E}^*$ be the non-parametric trace in \mathcal{E}^* defined as:*

- $\epsilon \upharpoonright_\theta = \epsilon$, where ϵ is the empty trace/word, and
- $(\tau e \langle \theta' \rangle) \upharpoonright_\theta = \begin{cases} (\tau \upharpoonright_\theta) e & \text{when } \theta' \sqsubseteq \theta \\ \tau \upharpoonright_\theta & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$

Therefore, the trace slice $\tau \upharpoonright_\theta$ first filters out all the parametric events that are not relevant for the instance θ , i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ , it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard parameter instances that are not relevant to θ during the slicing, including those more informative than θ , in order to achieve a correct slice for θ : in our running example, the

trace slice for $\langle \rangle$ should contain only begin and end events and no acquire or release. Otherwise, the acquire and release of different resources will interfere with each other in the trace slice for $\langle \rangle$.

One should not confuse extracting/abstracting traces from executions with slicing traces. The former determines the events to include in the trace, as well as parameter instances carried by events, while the latter dispatches each event in the given trace to corresponding trace slices according to the event's parameter instance. Different abstractions may result in different parametric traces from the same execution and thus may lead to different trace slices for the same parameter instance θ . For the (map, collection, iterator) example in Section 17.1, $X = \{m, c, i\}$ and an execution may generate the following parametric trace: `createColl` $\langle m_1, c_1 \rangle$ `createIter` $\langle c_1, i_1 \rangle$ `next` $\langle i_1 \rangle$ `updateMap` $\langle m_1 \rangle$. The trace slice for $\langle m_1 \rangle$ is `updateMap` for this parametric trace. Now suppose that we are only interested in operations on maps. Then $X = \{m\}$ and the trace abstracted from the execution generating the above trace is `createColl` $\langle m_1 \rangle$ `updateMap` $\langle m_1 \rangle$, in which events and parameter bindings irrelevant to m are removed. Then the trace slice for $\langle m_1 \rangle$ is `createColl` `updateMap`. In this paper we focus only on trace slicing; more discussion about trace abstraction can be found in [64].

Specifying properties over parametric traces is rather challenging, because one may want to specify a property for one generic parameter instance and then say “and so on for all the other instances”. In other words, one may want to specify a sort of a universally quantified property over base events, but, unfortunately, the underlying specification formalism may not allow such quantification over data; for example, none of the conventional formalisms to specify properties on linear traces listed above (i.e, linear temporal logics, regular expressions, context-free grammars) or mentioned in the rest of the paper has data quantification by default. We say “by default” because, in some cases, there are attempts to add data quantification; for example, [185] investigates the implications and the necessary restrictions resulting from adding quantifiers to LTL, and [257] investigates a finite-trace variant of parametric LTL together with a translation to parametric alternating automata.

Definition 75 *Let X be a set of parameters together with their corresponding parameter values V , like in Definition 102, and let $P : \mathcal{E}^* \rightarrow \mathcal{C}$ be a non-parametric property like in Definition 101. Then we define the **parametric property** $\lambda X . P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and categories $[[X \rightarrow$*

$V] \rightarrow \mathcal{C}]$)

$$\Lambda X . P \quad : \quad \mathcal{E}\langle X \rangle^* \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$$

defined as

$$(\Lambda X . P)(\tau)(\theta) = P(\tau \upharpoonright_{\theta})$$

for any $\tau \in \mathcal{E}\langle X \rangle^*$ and any $\theta \in [X \rightarrow V]$. If $X = \{x_1, \dots, x_n\}$ we may write $\Lambda x_1, \dots, x_n . P$ instead of $\Lambda\{x_1, \dots, x_n\} . P$. Also, if P_{φ} is defined using a pattern or formula φ in some particular trace specification formalism, we take the liberty to write $\Lambda X . \varphi$ instead of $\Lambda X . P_{\varphi}$.

justify why we want partial $[X \rightarrow V]$ instead of total in the semantics of the property itself above

Parametric properties $\Lambda X . P$ over base properties $P : \mathcal{E}^* \rightarrow \mathcal{C}$ are therefore properties taking traces in $\mathcal{E}\langle X \rangle^*$ to categories $[[X \rightarrow V] \rightarrow \mathcal{C}]$, i.e., to function domains from parameter instances to base property categories. $\Lambda X . P$ is defined as if many instances of P are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

Example 25 (part 5) *Let P be the non-parametric property specified by the regular expression in the second part of our running example above (using the mapping of regular expressions to properties discussed in the second part of our running example and after Definition 101 – i.e., the JavaMOP/RV semantic approach to parametric monitoring [62]). Since we want P to hold for any resource instance, we define the following parametric property:*

$$\Lambda r . (\text{begin } (\epsilon \mid (\text{acquire } (\text{acquire} \mid \text{release})^* \text{release})) \text{ end})^*.$$

If τ is the parametric trace and θ_1 and θ_2 are the parameter instances in the fourth part of our running example, then the semantics of the parametric property above on trace τ is validating for parameter instance θ_1 and violating for parameter instance θ_2 .

15.5 Slicing With Less

Consider a parametric trace τ in $\mathcal{E}\langle X \rangle^*$ and a parameter instance θ . Since there is no apriori correlation between the parameters being instantiated

by θ and those by the various parametric events in τ , it may very well be the case that θ contains parameter instances that never appear in τ . In this section we show that slicing τ by θ is the same as slicing it by a “smaller” parameter instance than θ , namely one containing only those parameters instantiated by θ that also appear as instances of some parameters in some events in τ . Formally, this smaller parameter instance is the largest partial map smaller than θ in the lub closure of all the parameter instances of events in τ ; this partial function is proved to indeed exist. We first formalize a notation used informally so far in this paper:

Notation 1 *When the domain of θ is finite, which is always the case in our examples in this paper and probably will also be the case in most practical uses of our trace slicing algorithm, and when the corresponding parameter names are clear from context, we take the liberty to write partial functions compactly by simply listing their parameter values; for example, we write a partial function θ with $\theta(a) = a_2$, $\theta(b) = b_1$ and $\theta(c) = c_1$ as the sequence “ $a_2b_1c_1$ ”. The function \perp then corresponds to the empty sequence.*

Example 26 *Here is a parametric trace with events parametric in $\{a, b, c\}$, where the parameters take values in the set $\{a_1, a_2, b_1, c_1\}$:*

$$\tau = e_1\langle a_1 \rangle e_2\langle a_2 \rangle e_3\langle b_1 \rangle e_4\langle a_2b_1 \rangle e_5\langle a_1 \rangle e_6\langle \rangle e_7\langle b_1 \rangle e_8\langle c_1 \rangle e_9\langle a_2c_1 \rangle e_{10}\langle a_1b_1c_1 \rangle e_{11}\langle \rangle.$$

It may be the case that some of the base events appearing in a trace are the same; for example, e_1 may be equal to e_2 and to e_5 . It is actually frequently the case in practice (at least in PQL [192], Tracematches [9], JavaMOP [62] and RV [197]) that parametric events are specified apriori with a given (sub)set of parameters, so that each event in \mathcal{E} is always instantiated with partial functions over the same domain, i.e., if $e\langle\theta\rangle$ and $e\langle\theta'\rangle$ appear in a parametric trace, then $\text{Dom}(\theta) = \text{Dom}(\theta')$. While this restriction is reasonable and sometimes useful, our trace slicing and monitoring algorithms in this paper do not need it.

Recall from Definition 105 that the trace slice $\tau|_\theta$ keeps from τ only those events that are relevant for θ and drops their parameters.

Example 27 *Consider again the sample parametric trace above with events parametric in $\{a, b, c\}$: $\tau = e_1\langle a_1 \rangle e_2\langle a_2 \rangle e_3\langle b_1 \rangle e_4\langle a_2b_1 \rangle e_5\langle a_1 \rangle e_6\langle \rangle e_7\langle b_1 \rangle e_8\langle c_1 \rangle e_9\langle a_2c_1 \rangle e_{10}\langle a_1b_1c_1 \rangle e_{11}\langle \rangle$*

Several slices of τ are listed below:

$$\begin{aligned}
\tau|_{a_1} &= e_1 e_5 e_6 e_{11} \\
\tau|_{a_2} &= e_2 e_6 e_{11} \\
\tau|_{a_1 b_1} &= e_1 e_3 e_5 e_6 e_7 e_{11} \\
\tau|_{a_2 b_1} &= e_2 e_3 e_4 e_6 e_7 e_{11} \\
\tau| &= e_6 e_{11} \\
\tau|_{a_1 b_1 c_1} &= e_1 e_3 e_5 e_6 e_7 e_8 e_{10} e_{11} \\
\tau|_{a_2 b_1 c_1} &= e_2 e_3 e_4 e_6 e_7 e_8 e_9 e_{11} \\
\tau|_{a_1 b_2 c_1} &= e_1 e_5 e_6 e_8 e_{11} \\
\tau|_{b_2 c_2} &= e_6 e_{11}
\end{aligned}$$

In order for the partial functions above to make sense, we assumed that the set V in which parameters $X = \{a, b, c\}$ take values includes $\{a_1, a_2, b_1, c_1\}$.

Definition 76 Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$, we let Θ_τ denote the lub closure of all the parameter instances appearing in events in τ , that is, $\Theta_\tau = \overline{\{\theta \mid \theta \in [X \rightarrow V], e\langle\theta\rangle \in \tau\}}$.

Proposition 33 For any parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$, the set Θ_τ is a finite lub closed set.

Proof: Θ_τ is already defined as a lub closed set; since τ is finite, Corollary 6 implies that Θ_τ is finite. \square

Proposition 34 Given $\tau e\langle\theta\rangle \in \mathcal{E}\langle X \rangle^*$, the following equality holds: $\Theta_{\tau e\langle\theta\rangle} = \{\perp, \theta\} \sqcup \Theta_\tau$.

Proof: It follows by the following sequence of equalities:

$$\begin{aligned}
\Theta_{\tau e\langle\theta\rangle} &= \overline{\{\theta' \mid \theta' \in [X \rightarrow V], e'\langle\theta'\rangle \in \tau e\langle\theta\rangle\}} \\
&= \overline{\{\theta\} \cup \{\theta' \mid \theta' \in [X \rightarrow V], e'\langle\theta'\rangle \in \tau\}} \\
&= \overline{\{\theta\} \cup \Theta_\tau} \\
&= \{\perp, \theta\} \sqcup \overline{\Theta_\tau} \\
&= \{\perp, \theta\} \sqcup \Theta_\tau.
\end{aligned}$$

The first equality follows by Definition 76, the second by separating the case $e'\langle\theta'\rangle = e\langle\theta\rangle$, the third again by Definition 76, the fourth by Corollary 5, and the fifth by Proposition 33. Therefore, $\Theta_{\tau e\langle\theta\rangle}$ is the smallest lub closed set that contains θ and includes Θ_τ . \square

Proposition 35 *Given $\tau \in \mathcal{E}\langle X \rangle^*$ and $\theta \in [X \rightarrow V]$, the equality $\tau|_{\theta} = \tau|_{\max(\theta)|_{\Theta_{\tau}}}$ holds.*

Proof: We prove the following more general result:

“Let $\Theta \subseteq [X \rightarrow V]$ be lub closed and let $\theta \in [X \rightarrow V]$;

then $\tau|_{\theta} = \tau|_{\max(\theta)|_{\Theta}}$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ with $\Theta_{\tau} \subseteq \Theta$.”

First note that the statement above is well-formed because $\max(\theta)|_{\Theta}$ exists whenever Θ is lub closed (1. in Proposition 29), and that it is indeed more general than the stated result: for the given $\tau \in \mathcal{E}\langle X \rangle^*$ and $\theta \in [X \rightarrow V]$, we pick Θ to be Θ_{τ} . We prove the general result by induction on the *length* of τ :

- If $|\tau| = 0$ then $\tau = \epsilon$ and $\epsilon|_{\theta} = \epsilon|_{\max(\theta)|_{\Theta}} = \epsilon$.

- Now suppose that $\tau|_{\theta} = \tau|_{\max(\theta)|_{\Theta}}$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ with $\Theta_{\tau} \subseteq \Theta$ and $|\tau| = n \geq 0$, and let us show that $\tau'|_{\theta} = \tau'|_{\max(\theta)|_{\Theta}}$ for any $\tau' \in \mathcal{E}\langle X \rangle^*$ with $\Theta_{\tau'} \subseteq \Theta$ and $|\tau'| = n + 1$. Pick such a τ' and let $\tau' = \tau e\langle \theta' \rangle$ for a $\tau \in \mathcal{E}\langle X \rangle^*$ with $|\tau| = n$ and an $e\langle \theta' \rangle \in \mathcal{E}\langle X \rangle$. Since $\Theta_{\tau'} \subseteq \Theta$, by 6. in Proposition 25 and by Proposition 34 it follows that $\Theta_{\tau} \subseteq \{\perp, \theta'\} \sqcup \Theta_{\tau} \subseteq \Theta$, so the induction hypothesis implies $\tau|_{\theta} = \tau|_{\max(\theta)|_{\Theta}}$. The rest follows noticing that $\theta' \sqsubseteq \theta$ iff $\theta' \sqsubseteq \max(\theta)|_{\Theta}$, which is a consequence of the definition of $\max(\theta)|_{\Theta}$ because $\theta' \in \{\perp, \theta'\} \subseteq \{\perp, \theta'\} \sqcup \Theta_{\tau} \subseteq \Theta$ (again by 6. in Proposition 25 and by Proposition 34).

Alternatively, one could have also done the proof above by induction on τ , not on its length, but the proof would be more involved, because one would need to prove that the domain over which the property is universally quantified, namely “any $\tau \in \mathcal{E}\langle X \rangle^*$ with $\Theta_{\tau} \subseteq \Theta$ ” is inductively generated. We therefore preferred to choose a more elementary induction schema. \square

15.6 Algorithm for Online Parametric Trace Slicing

Definition 105 illustrates a way to slice a parametric trace for *given* parameter bindings. However, it is not suitable for online trace slicing, where the trace is observed incrementally and no future knowledge is available, because we cannot know all possible parameter instances θ apriori. We next define an algorithm $\mathbb{A}\langle X \rangle$ that takes a parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ incrementally (i.e., event by event), and builds a partial function $\mathbb{T} \in [[X \rightarrow V] \rightarrow \mathcal{E}^*]$ of finite domain that serves as a quick lookup table for all slices of τ . More

Algorithm $\mathbb{A}\langle X \rangle$
Input: parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$
Output: map $\mathbb{T} \in [[X \rightarrow V] \rightarrow \mathcal{E}^*]$ and set $\Theta \subseteq [X \rightarrow V]$

```

1  $\mathbb{T} \leftarrow \perp$ ;  $\mathbb{T}(\perp) \leftarrow \epsilon$ ;  $\Theta \leftarrow \{\perp\}$ 
2 foreach parametric event  $e\langle\theta\rangle$  in order (first to last) in  $\tau$  do
3   foreach  $\theta' \in \{\theta\} \sqcup \Theta$  do
4      $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\max(\theta')_\Theta) e$ 
5   endfor
6    $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$ 
7 endfor

```

Figure 15.2: Parametric trace slicing algorithm $\mathbb{A}\langle X \rangle$.

precisely, Theorem 23 shows that, for any $\theta \in [X \rightarrow V]$, the trace slice $\tau|_\theta$ is $\mathbb{T}(\max(\theta)_\Theta)$ after $\mathbb{A}\langle X \rangle$ processes τ , where $\Theta = \Theta_\tau$ is the domain of \mathbb{T} , a finite lub closed set of partial functions also calculated by $\mathbb{A}\langle X \rangle$ incrementally (see Definition 76 for Θ_τ). Therefore, assuming that $\mathbb{A}\langle X \rangle$ is run on trace τ , all one has to do in order to calculate a slice $\tau|_\theta$ for a given $\theta \in [X \rightarrow V]$ is to calculate $\max(\theta)_\Theta$ followed by a lookup into \mathbb{T} . This way the trace τ , which can be very long, is processed/traversed only once, as it is being generated, and appropriate data-structures are maintained by our algorithm that allow for retrieval of slices for any parameter instance θ , without having to traverse the trace τ again, as an algorithm blindly following the definition of trace slicing (Definition 105) would do.

Figure 15.2 shows our trace slicing algorithm $\mathbb{A}\langle X \rangle$. In spite of $\mathbb{A}\langle X \rangle$'s small size, its proof of correctness is surprisingly intricate, making use of almost all the mathematical machinery developed so far in the paper. The algorithm $\mathbb{A}\langle X \rangle$ on input τ , written more succinctly $\mathbb{A}\langle X \rangle(\tau)$, traverses τ from its first event to its last event and, for each encountered event $e\langle\theta\rangle$, updates both its data-structures, \mathbb{T} and Θ . After processing each event, the relationship between \mathbb{T} and Θ is that the latter is the domain of the former. Line 1 initializes the data-structures: \mathbb{T} is undefined everywhere (i.e., \perp) except for the undefined-everywhere function \perp , where $\mathbb{T}(\perp) = \epsilon$; as expected, Θ is then initialized to the set $\{\perp\}$. The code (lines 3 to 6) inside the outer loop (lines 2 to 7) can be triggered when a new event is received, as in most online runtime verification systems. When a new

Example 28

$e_1\langle a_1 \rangle$	$e_2\langle a_2 \rangle$	$e_3\langle b_1 \rangle$	$e_4\langle a_2b_1 \rangle$	$e_5\langle a_1 \rangle$	$e_6\langle \rangle$	$e_7\langle b_1 \rangle$
$\langle \rangle: \epsilon$ $\langle a_1 \rangle: e_1$	$\langle \rangle: \epsilon$ $\langle a_1 \rangle: e_1$ $\langle a_2 \rangle: e_2$	$\langle \rangle: \epsilon$ $\langle a_1 \rangle: e_1$ $\langle a_2 \rangle: e_2$ $\langle b_1 \rangle: e_3$ $\langle a_1b_1 \rangle: e_1e_3$ $\langle a_2b_1 \rangle: e_2e_3$	$\langle \rangle: \epsilon$ $\langle a_1 \rangle: e_1$ $\langle a_2 \rangle: e_2$ $\langle b_1 \rangle: e_3$ $\langle a_1b_1 \rangle: e_1e_3$ $\langle a_2b_1 \rangle: e_2e_3e_4$	$\langle \rangle: \epsilon$ $\langle a_1 \rangle: e_1e_5$ $\langle a_2 \rangle: e_2$ $\langle b_1 \rangle: e_3$ $\langle a_1b_1 \rangle: e_1e_3e_5$ $\langle a_2b_1 \rangle: e_2e_3e_4$	$\langle \rangle: e_6$ $\langle a_1 \rangle: e_1e_5e_6$ $\langle a_2 \rangle: e_2e_6$ $\langle b_1 \rangle: e_3e_6$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6$	$\langle \rangle: e_6$ $\langle a_1 \rangle: e_1e_5e_6$ $\langle a_2 \rangle: e_2e_6$ $\langle b_1 \rangle: e_3e_6$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6$

$e_8\langle c_1 \rangle$	$e_9\langle a_2c_1 \rangle$	$e_{10}\langle a_1b_1c_1 \rangle$	$e_{11}\langle \rangle$
$\langle \rangle: e_6$ $\langle a_1 \rangle: e_1e_5e_6$ $\langle a_2 \rangle: e_2e_6$ $\langle b_1 \rangle: e_3e_6e_7$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6e_7$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6e_7$ $\langle c_1 \rangle: e_6e_8$ $\langle a_1c_1 \rangle: e_1e_5e_6e_8$ $\langle a_2c_1 \rangle: e_2e_6e_8$ $\langle b_1c_1 \rangle: e_3e_6e_7e_8$ $\langle a_1b_1c_1 \rangle: e_1e_3e_5e_6e_7e_8$ $\langle a_2b_1c_1 \rangle: e_2e_3e_4e_6e_7e_8$	$\langle \rangle: e_6$ $\langle a_1 \rangle: e_1e_5e_6$ $\langle a_2 \rangle: e_2e_6$ $\langle b_1 \rangle: e_3e_6e_7$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6e_7$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6e_7$ $\langle c_1 \rangle: e_6e_8$ $\langle a_1c_1 \rangle: e_1e_5e_6e_8$ $\langle a_2c_1 \rangle: e_2e_6e_8e_9$ $\langle b_1c_1 \rangle: e_3e_6e_7e_8$ $\langle a_1b_1c_1 \rangle: e_1e_3e_5e_6e_7e_8$ $\langle a_2b_1c_1 \rangle: e_2e_3e_4e_6e_7e_8e_9$	$\langle \rangle: e_6$ $\langle a_1 \rangle: e_1e_5e_6$ $\langle a_2 \rangle: e_2e_6$ $\langle b_1 \rangle: e_3e_6e_7$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6e_7$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6e_7$ $\langle c_1 \rangle: e_6e_8$ $\langle a_1c_1 \rangle: e_1e_5e_6e_8$ $\langle a_2c_1 \rangle: e_2e_6e_8e_9$ $\langle b_1c_1 \rangle: e_3e_6e_7e_8$ $\langle a_1b_1c_1 \rangle: e_1e_3e_5e_6e_7e_8e_{10}$ $\langle a_2b_1c_1 \rangle: e_2e_3e_4e_6e_7e_8e_9$	$\langle \rangle: e_6e_{11}$ $\langle a_1 \rangle: e_1e_5e_6e_{11}$ $\langle a_2 \rangle: e_2e_6e_{11}$ $\langle b_1 \rangle: e_3e_6e_7e_{11}$ $\langle a_1b_1 \rangle: e_1e_3e_5e_6e_7e_{11}$ $\langle a_2b_1 \rangle: e_2e_3e_4e_6e_7e_{11}$ $\langle c_1 \rangle: e_6e_8e_{11}$ $\langle a_1c_1 \rangle: e_1e_5e_6e_8e_{11}$ $\langle a_2c_1 \rangle: e_2e_6e_8e_9e_{11}$ $\langle b_1c_1 \rangle: e_3e_6e_7e_8e_{11}$ $\langle a_1b_1c_1 \rangle: e_1e_3e_5e_6e_7e_8e_{10}e_{11}$ $\langle a_2b_1c_1 \rangle: e_2e_3e_4e_6e_7e_8e_9e_{11}$

Table 15.1: A run of the trace slicing algorithm $\mathbb{A}\langle X \rangle$ (top-left table first, followed by bottom-left table, followed by the right table).

event is received, say $e\langle \theta \rangle$, the mapping \mathbb{T} is updated as follows: for each $\theta' \in [X \rightarrow V]$ that can be obtained by combining θ with the compatible partial functions in the domain of the current \mathbb{T} , update $\mathbb{T}(\theta')$ by adding the non-parametric event e to the end of the slice corresponding to the largest (i.e., most “knowledgeable”) entry in the current table \mathbb{T} that is less informative or as informative as θ' ; the Θ data-structure is then extended in line 6 (see Proposition 34 for why this way).

Consider again the sample trace in Section 15.5 with events parametric in $\{a, b, c\}$, namely $\tau = e_1\langle a_1 \rangle e_2\langle a_2 \rangle e_3\langle b_1 \rangle e_4\langle a_2b_1 \rangle e_5\langle a_1 \rangle e_6\langle \rangle e_7\langle b_1 \rangle e_8\langle c_1 \rangle e_9\langle a_2c_1 \rangle e_{10}\langle a_1b_1c_1 \rangle e_{11}\langle \rangle$. Table 15.1 shows how $\mathbb{A}\langle X \rangle$ works on τ . An entry of the form $\langle \theta \rangle: w$ in a table cell corresponding to a current parametric event $e\langle \theta \rangle$ means that $\mathbb{T}(\theta) = w$ after processing all the parametric events up to and including the current one; \mathbb{T} is undefined on any other partial function. Obviously, the Θ corresponding to a cell is the union of all the θ ’s that appear in pairs $\langle \theta \rangle: w$ in that cell. Note that, as each parametric event $e\langle \theta \rangle$ is processed, the

non-parametric event e is added at most once to each slice, and that Θ stays lub closed.

$\mathbb{A}\langle X \rangle$ computes trace slices for all combinations of parameter instances observed in parametric trace events. Its complexity is therefore $O(n \times m)$ where n is the length of the trace and m is the number of all possible parameter combinations. However, $\mathbb{A}\langle X \rangle$ is not intended to be implemented directly; it is only used as a correctness backbone for other trace analysis algorithms, such as the monitoring algorithms discussed below. An alternative and apparently more efficient solution is to only record trace slices for parameter instances that actually appear in the trace (instead of for all combinations of them), and then construct the slice for a given parameter instance by combining such trace slices for compatible parameter instances. However, the complexity of constructing all possible trace slices at the end using such an algorithm is also $O(n \times m)$, so it would not bring any benefit overall compared to $\mathbb{A}\langle X \rangle$. In addition, $\mathbb{A}\langle X \rangle$ is more suitable as a backbone for developing online monitoring algorithms such as those in Section 15.7, because each event is sent to its slices (that will be consumed by corresponding monitors) and never touched again.

$\mathbb{A}\langle X \rangle$ compactly and uniformly captures several special cases and subcases that are worth discussing. The discussion below can be formalized as an inductive (on the length of τ) proof of correctness for $\mathbb{A}\langle X \rangle$, but we prefer to keep this discussion informal and give a rigorous proof shortly after. The role of this discussion is twofold: (1) to better explain the algorithm $\mathbb{A}\langle X \rangle$, providing the reader with additional intuition for its difficulty and compactness, and (2) to give a proof sketch for the correctness of $\mathbb{A}\langle X \rangle$.

Let us first note that a partial function added to Θ will never be removed from Θ ; that's because $\Theta \subseteq \{\perp, \theta\} \sqcup \Theta$. The same holds true for the domain of \mathbb{T} , because line 4 can only add new elements to $\text{Dom}(\mathbb{T})$; in fact, the domain of \mathbb{T} is extended with precisely the set $\{\theta\} \sqcup \Theta$ after each event parametric in θ is processed by $\mathbb{A}\langle X \rangle$. Moreover, since $\text{Dom}(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\perp\}$ initially and since 5. and 7. in Proposition 25 imply $\Theta \cup (\{\theta\} \sqcup \Theta) = \{\perp, \theta\} \sqcup \Theta$ while Proposition 34 states that $\Theta_{\tau e(\theta)} = \{\perp, \theta\} \sqcup \Theta_\tau$, we can inductively show that $\text{Dom}(\mathbb{T}) = \Theta = \Theta_\tau$ each time after $\mathbb{A}\langle X \rangle$ is executed on a parametric trace τ .

Each θ' considered by the loop at lines 3-5 has the property that $\theta \sqsubseteq \theta'$, and at (precisely) one iteration of the loop θ' is θ ; indeed, $\theta \in \{\theta\} \sqcup \Theta$ because $\perp \in \Theta$. Thanks to Proposition 35, Theorem 23 holds essentially iff $\mathbb{T}(\theta') = \tau|_{\theta'}$ after $\mathbb{T}(\theta')$ is updated in line 4. A tricky observation which is

crucial for this is that β . in Proposition 29 implies that the updates of $\mathbb{T}(\theta')$ do not interfere with each other for different $\theta' \in \{\theta\} \sqcup \Theta$; otherwise the non-parametric event e may wrongly be added multiple times to some trace slices $\mathbb{T}(\theta')$.

Let us next informally argue, inductively, that it is indeed the case that $\mathbb{T}(\theta') = \tau \upharpoonright_{\theta'}$ after $\mathbb{T}(\theta')$ is updated in line 4 (it vacuously holds on the empty trace). Since $\max(\theta')_{\Theta} \in \Theta$, the inductive hypothesis tells us that $\mathbb{T}(\max(\theta')_{\Theta}) = \tau \upharpoonright_{\max(\theta')_{\Theta}}$; these are further equal to $\tau \upharpoonright_{\theta'}$ by Proposition 35. Since $\theta \sqsubseteq \theta'$, the definition of trace slicing implies that $(\tau e\langle\theta\rangle) \upharpoonright_{\theta'} = \tau \upharpoonright_{\theta'} e$. Therefore, $\mathbb{T}(\theta')$ is indeed $(\tau e\langle\theta\rangle) \upharpoonright_{\theta'}$ after line 4 of $\mathbb{A}\langle X \rangle$ is executed while processing the event $e\langle\theta\rangle$ that follows trace τ . This concludes our informal proof sketch; let us next give a rigorous proof of correctness for our trace slicing algorithm $\mathbb{A}\langle X \rangle$.

Definition 77 Let $\mathbb{A}\langle X \rangle(\tau).\mathbb{T}$ and $\mathbb{A}\langle X \rangle(\tau).\Theta$ be the two data-structures (\mathbb{T} and Θ) maintained by the algorithm $\mathbb{A}\langle X \rangle$ in Figure 15.2 after it processes τ .

Theorem 23 With the notation in Definition 77, the following hold for any $\tau \in \mathcal{E}\langle X \rangle^*$:

1. $\text{Dom}(\mathbb{A}\langle X \rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X \rangle(\tau).\Theta = \Theta_{\tau}$;
2. $\mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\theta) = \tau \upharpoonright_{\theta}$ for any $\theta \in \mathbb{A}\langle X \rangle(\tau).\Theta$;
3. $\tau \upharpoonright_{\theta} = \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\max(\theta)_{\mathbb{A}\langle X \rangle(\tau).\Theta})$ for any $\theta \in [X \rightarrow V]$.

Proof: Since $\mathbb{A}\langle X \rangle$ processes the events in the input trace in order, when given the input $\tau e\langle\theta\rangle$, the Θ and \mathbb{T} structures after $\mathbb{A}\langle X \rangle$ processes τ but before it processes $e\langle\theta\rangle$ (i.e., right before the last iteration of the loop at lines 2-7) are precisely $\mathbb{A}\langle X \rangle(\tau).\Theta$ and $\mathbb{A}\langle X \rangle(\tau).\mathbb{T}$, respectively. Further, the loop at lines 3-5 updates \mathbb{T} on all $\theta' \in \{\theta\} \sqcup \Theta$; in case \mathbb{T} was not defined on such a θ' , then it will be defined after $e\langle\theta\rangle$ is processed. The definitional domain of \mathbb{T} is thus continuously growing or potentially remains stationary as parametric events are processed, but it never decreases. With these observations, we can prove 1. by induction on τ . If $\tau = \epsilon$ then $\text{Dom}(\mathbb{A}\langle X \rangle(\epsilon).\mathbb{T}) = \mathbb{A}\langle X \rangle(\epsilon).\Theta = \Theta_{\epsilon} = \{\perp\}$. Suppose now that $\text{Dom}(\mathbb{A}\langle X \rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X \rangle(\tau).\Theta = \Theta_{\tau}$ holds for $\tau \in \mathcal{E}\langle X \rangle^*$, and let $e\langle\theta\rangle \in \mathcal{E}\langle X \rangle$ be any parametric event. Then the following

concludes the proof of 1.:

$$\begin{aligned}
\text{Dom}(\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}) &= \text{Dom}(\mathbb{A}\langle X \rangle(\tau).\mathbb{T}) \cup (\{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta) \\
&= \mathbb{A}\langle X \rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta) \\
&= (\{\perp\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta) \cup (\{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta) \\
&= \{\perp, \theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta \\
&= \mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\Theta \\
&= \{\perp, \theta\} \sqcup \Theta_\tau \\
&= \Theta_{\tau e\langle \theta \rangle}
\end{aligned}$$

where the first equality follows from how the loop at lines 3-5 updates \mathbb{T} , the second by the induction hypothesis, the third by 5. in Proposition 25, the fourth by 7. in Proposition 25, the fifth by how Θ is updated at line 6, the sixth again by the induction hypothesis, and, finally, the seventh by Proposition 34.

Before we continue, let us first prove the following property:

$$\begin{aligned}
&\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\max(\theta']_{\mathbb{A}\langle X \rangle(\tau).\Theta}) e \\
&\text{for any } e\langle \theta \rangle \in \mathcal{E}\langle X \rangle \text{ and any } \theta' \in \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta.
\end{aligned}$$

One should be careful here to *not* get tricked thinking that this property is straightforward, because it says only what line 4 of $\mathbb{A}\langle X \rangle$ does. The complexity comes from the fact that if there were two different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$ such that $\theta_1 = \max(\theta_2]_{\mathbb{A}\langle X \rangle(\tau).\Theta}$, then an unfortunate enumeration of the partial functions θ' in $\{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$ by the loop at lines 3-5 may lead to the non-parametric event e to be added twice to a slice: indeed, if θ_1 is processed before θ_2 , then e is first added to the end of $\mathbb{T}(\theta_1)$ when $\theta' = \theta_1$, and then $\mathbb{T}(\theta_1)e$ is assigned to $\mathbb{T}(\theta_2)$ when $\theta' = \theta_2$; this way, $\mathbb{T}(\theta_2)$ ends up accumulating e twice instead of once, which is obviously wrong. Fortunately, since $\mathbb{A}\langle X \rangle(\tau).\Theta$ is lub closed (by 1. above and Proposition 33), 3. in Proposition 29 implies that there are no such different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$. Therefore, there is no interference between the various assignments at line 4, regardless of the order in which the partial functions $\theta' \in \{\theta\} \sqcup \Theta$ are enumerated, which means that, indeed, $\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\max(\theta']_{\mathbb{A}\langle X \rangle(\tau).\Theta}) e$ for any $e\langle \theta \rangle \in \mathcal{E}\langle X \rangle$ and for any $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$. This lack of interference between updates of \mathbb{T} also suggests an important implementation optimization:

The loop at lines 3-5 can be parallelized without duplicating the table \mathbb{T} !

Of course, the loop can be parallelized anyway if the table is duplicated and then merged within the original table, in the sense that all the writes to $\mathbb{T}(\theta')$ are done in a copy of \mathbb{T} . However, experiments show that the table \mathbb{T} can be literally huge in real applications, in the order of billions of entries, so duplicating and merging it can be prohibitive.

2. can be now proved by induction on the length of τ . If $\tau = \epsilon$ then $\mathbb{A}\langle X \rangle(\epsilon).\Theta = \{\perp\}$, so $\theta' \in \mathbb{A}\langle X \rangle(\epsilon).\Theta$ can only be \perp ; then $\mathbb{A}\langle X \rangle(\epsilon).\mathbb{T}(\perp) = \tau \upharpoonright_{\perp} = \epsilon$. Suppose now that $\mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\theta') = \tau \upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X \rangle(\tau).\Theta$ and let us show that $\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') = (\tau e\langle \theta \rangle) \upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\Theta$. As shown in the proof of 1. above, $\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\Theta = \mathbb{A}\langle X \rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta)$, so we have two cases to analyze. First, if $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$ then $\theta \sqsubseteq \theta'$ and so $(\tau e\langle \theta \rangle) \upharpoonright_{\theta'} = \tau \upharpoonright_{\theta'} e$; further,

$$\begin{aligned} \mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') &= \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\max(\theta')_{\mathbb{A}\langle X \rangle(\tau).\Theta}) e \\ &= \tau \upharpoonright_{\max(\theta')_{\mathbb{A}\langle X \rangle(\tau).\Theta}} e \\ &= \tau \upharpoonright_{\theta'} e \\ &= (\tau e\langle \theta \rangle) \upharpoonright_{\theta'}, \end{aligned}$$

where the first equality follows by the auxiliary property proved above, the second by the induction hypothesis using the fact that $\max(\theta')_{\mathbb{A}\langle X \rangle(\tau).\Theta} \in \mathbb{A}\langle X \rangle(\tau).\Theta$, and the third by Proposition 35. Second, if $\theta' \in \mathbb{A}\langle X \rangle(\tau).\Theta$ but $\theta' \not\sqsubseteq \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$ then $\theta \not\sqsubseteq \theta'$ and so $(\tau e\langle \theta \rangle) \upharpoonright_{\theta'} = \tau \upharpoonright_{\theta'}$; furthermore,

$$\begin{aligned} \mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') &= \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\theta') \\ &= \tau \upharpoonright_{\theta'} \\ &= (\tau e\langle \theta \rangle) \upharpoonright_{\theta'}, \end{aligned}$$

where the first equality holds because θ' is not considered by the loop in lines 3-5 in $\mathbb{A}\langle X \rangle$, that is, $\theta' \not\sqsubseteq \{\theta\} \sqcup \mathbb{A}\langle X \rangle(\tau).\Theta$, and the second equality follows by the induction hypothesis, as $\theta' \in \mathbb{A}\langle X \rangle(\tau).\Theta$. Therefore, $\mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\mathbb{T}(\theta') = (\tau e\langle \theta \rangle) \upharpoonright_{\theta'}$ for any $\theta' \in \mathbb{A}\langle X \rangle(\tau e\langle \theta \rangle).\Theta$, which completes the proof of 2.

3. is the main result concerning our trace slicing algorithm and it follows now easily:

$$\begin{aligned} \tau \upharpoonright_{\theta} &= \tau \upharpoonright_{\max(\theta)_{\Theta_{\tau}}} \\ &= \tau \upharpoonright_{\max(\theta)_{\mathbb{A}\langle X \rangle(\tau).\Theta}} \\ &= \mathbb{A}\langle X \rangle(\tau).\mathbb{T}(\max(\theta)_{\mathbb{A}\langle X \rangle(\tau).\Theta}) \end{aligned}$$

The first equality follows by Proposition 35, the second equality by 1. above, and the third equality by 2. above, because $\max(\theta)_{\mathbb{A}\langle X \rangle(\tau).\Theta} \in \mathbb{A}\langle X \rangle(\tau).\Theta$. This concludes the correctness proof of our trace slicing algorithm $\mathbb{A}\langle X \rangle$. \square

15.7 Monitors and Parametric Monitors

In this section we first define monitors M as a variant of Moore machines with potentially infinitely many states; then we define parametric monitors $\Lambda X.M$ as monitors maintaining one state of M per parameter instance. Like for parametric properties, which turned out to be just properties over parametric traces, we show that parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories. We also show that a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with P the property monitored by M .

15.7.1 The Non-Parametric Case

We start by defining non-parametric monitors as a variant of (deterministic) Moore machine [202] that allows infinitely many states:

most of this section should go to Chapter 4

Definition 78 A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, \iota, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, where S is a set of states, \mathcal{E} is a set of input events, \mathcal{C} is a set of output categories, $\iota \in S$ is the initial state, σ is the transition function, and γ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \rightarrow S$ as expected: $\sigma(s, \epsilon) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in \mathcal{E}$, and $w \in \mathcal{E}^*$.

The notion of a monitor above is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a “by need” basis. Consider, for example, a monitor for a property specified using an NFA which performs an NFA-to-DFA construction on the fly, as events are received. Such a monitor generates only those states in the DFA that are needed by the monitored execution trace. Moreover, the monitor only needs to store one such state of the DFA, i.e., set of states in the NFA, namely the current one: once an event is received, the next state is (deterministically) computed and the old one is discarded. Therefore, assuming that one needs constant space to store a state of the original NFA, then the memory needed by this monitor is linear in the number of states of the NFA. An alternative and probably more conventional monitor could be one which generates the corresponding DFA statically, paying upfront the exponential price in both time and space. As empirically suggested by [261], if one is able to statically

generate and store the corresponding DFA then one should most likely take this route, because in practice it tends to be much faster to jump to a known next state than to compute it.

Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many states are reached. For example, monitors for context-free grammars like the ones in [195] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well. Nevertheless, what is common to all monitors is that they can classify traces into categories. When a monitor does not have enough information about a trace to put it in a category of interest, we can assume that it actually categorizes it as a “don’t know” trace, where “don’t know” can be regarded as a special category; this is similar to regarding partial functions as total functions by adding a special “undefined” value in their codomain. The following is therefore natural:

Definition 79 *Monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \rightarrow \mathcal{C}$ if and only if $\gamma(\sigma(\iota, w)) = P(w)$ for each $w \in \mathcal{E}^*$.*

A property can be associated to each monitor, in a similar style to which we can associate a language to each automaton:

Definition 80 *Monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$ defines **the M -property** $\mathcal{P}_M : \mathcal{E}^* \rightarrow \mathcal{C}$ as follows: $\mathcal{P}_M(w) = \gamma(\sigma(\iota, w))$ for each $w \in \mathcal{E}^*$.*

The following result is straightforward, it follows immediately from Definitions 79 and 80. The only reason we frame it as a numbered proposition is because we need to refer to it in the proof of Corollary 8.

Proposition 36 *With the notation in Definition 80, monitor M is indeed a monitor for its corresponding M -property \mathcal{P}_M . Moreover, a monitor can only be a monitor for one property, that is, if M is a monitor for property P then $P = \mathcal{P}_M$.*

Since we allow monitors to have infinitely many states, there is a strong correspondence between properties and monitors:

Definition 81 *Property $P : \mathcal{E}^* \rightarrow \mathcal{C}$ defines **the P -monitor** $\mathcal{M}_P = (S_P, \mathcal{E}, \mathcal{C}, \iota_P, \sigma_P, \gamma_P)$ as follows:*

$$S_P = \mathcal{E}^*,$$

$$\iota_P = \epsilon,$$

$$\sigma_P(w, e) = we \text{ for each } w \in S_P = \mathcal{E}^* \text{ and } e \in \mathcal{E},$$

$$\gamma_P(w) = P(w) \text{ for each } w \in S_P = \mathcal{E}^*.$$

Thus, \mathcal{M}_P holds traces as states, appends events to them as transition and, as output, it looks up the category of the corresponding trace using P . The following results are also straightforward and, again, we frame them as numbered propositions only because we will refer to them later.

Proposition 37 *With the notation in Definition 81, the monitor \mathcal{M}_P is indeed a monitor for property P .*

Proof: It follows from the sequence of equalities $\gamma_P(\sigma_P(\iota_P, w)) = \gamma_P(\sigma_P(\epsilon, w)) = \gamma_P(\epsilon w) = \gamma_P(w) = P(w)$. \square

Proposition 38 *With the notations in Definitions 80 and 81, $\mathcal{P}_{\mathcal{M}_P} = P$ for any property $P : \mathcal{E}^* \rightarrow \mathcal{C}$.*

Proof: $\mathcal{P}_{\mathcal{M}_P}(w) = \gamma_P(\sigma_P(\iota_P, w)) = P(w)$ for any $w \in \mathcal{E}^*$. \square

The equality of monitors $\mathcal{M}_{\mathcal{P}_M} = M$ does not hold for any monitor M ; it does hold when $M = \mathcal{M}_P$ for some property P , though.

Definition 82 *Monitors M and M' are **property equivalent**, or just **equivalent**, written $M \equiv M'$, iff they are monitors for the same property (see Definition 79). With the notation in Definition 80, we have that $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.*

Proposition 39 *With the notations in Definitions 80 and 81, $\mathcal{M}_{\mathcal{P}_M} \equiv M$ for any monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma)$.*

Proof: By Definition 82, $\mathcal{M}_{\mathcal{P}_M} \equiv M$ iff $\mathcal{P}_{\mathcal{M}_{\mathcal{P}_M}} = \mathcal{P}_M$, and the latter follows by Proposition 37 taking P to be \mathcal{P}_M . \square

15.7.2 The Parametric Case

We next define parametric monitors in the same style as the other parametric entities defined in this paper: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 83 *Given parameter set X with corresponding values V and a monitor $M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, we define the **parametric monitor** $\Lambda X.M$ as the monitor*

$$([X \rightarrow V] \rightarrow S], \mathcal{E}\langle X \rangle, [[X \rightarrow V] \rightarrow \mathcal{C}], \lambda\theta.\iota, \Lambda X.\sigma, \Lambda X.\gamma),$$

with

$$\begin{aligned} \Lambda X.\sigma &: [[X \rightarrow V] \rightarrow S] \times \mathcal{E}\langle X \rangle \rightarrow [[X \rightarrow V] \rightarrow S] \\ \Lambda X.\gamma &: [[X \rightarrow V] \rightarrow S] \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}] \end{aligned}$$

defined as

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$

$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for any $\delta \in [[X \rightarrow V] \rightarrow S]$ and any $\theta, \theta' \in [X \rightarrow V]$.

Therefore, a state δ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ , takes parametric events as input, and outputs categories indexed by parameter instances (one output category of M per parameter instance).

Proposition 40 *If M is a monitor for P then parametric monitor $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$, or, with the notation in Definition 80, $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$.*

Proof: We show that $(\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.\iota, \tau)) = (\Lambda X.P)(\tau)$ for any $\tau \in \mathcal{E}\langle X \rangle^*$, i.e., after application on $\theta \in [X \rightarrow V]$, that $\gamma((\Lambda X.\sigma)(\lambda\theta.\iota, \tau)(\theta)) = P(\tau \upharpoonright_\theta)$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ and $\theta \in [X \rightarrow V]$. Since M is a monitor for P , it suffices to show that $(\Lambda X.\sigma)(\lambda\theta.\iota, \tau)(\theta) = \sigma(\iota, \tau \upharpoonright_\theta)$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ and $\theta \in [X \rightarrow V]$. We prove it by induction on τ . If $\tau = \epsilon$ then $(\Lambda X.\sigma)(\lambda\theta.\iota, \epsilon)(\theta) = (\lambda\theta.\iota)(\theta) = \iota = \sigma(\iota, \epsilon) = \sigma(\iota, \epsilon \upharpoonright_\theta)$. Suppose that $(\Lambda X.\sigma)(\lambda\theta.\iota, \tau)(\theta) = \sigma(\iota, \tau \upharpoonright_\theta)$ for some arbitrary but fixed $\tau \in \mathcal{E}\langle X \rangle^*$ and

for any $\theta \in [X \rightarrow V]$, and let $e\langle\theta'\rangle$ be any parametric event in $\mathcal{E}\langle X \rangle$ and let $\theta \in [X \rightarrow V]$ be any parameter instance. The inductive step is then as follows:

$$\begin{aligned}
(\Lambda X . \sigma)(\lambda\theta.\iota, \tau e\langle\theta'\rangle)(\theta) &= (\Lambda X . \sigma)((\Lambda X . \sigma)(\lambda\theta.\iota, \tau), e\langle\theta'\rangle)(\theta) \\
&= (\Lambda X . \sigma)(\sigma(\iota, \tau\upharpoonright_\theta), e\langle\theta'\rangle)(\theta) \\
&= \begin{cases} \sigma(\sigma(\iota, \tau\upharpoonright_\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\iota, \tau\upharpoonright_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\
&= \begin{cases} \sigma(\iota, \tau\upharpoonright_\theta e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(\iota, \tau\upharpoonright_\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\
&= \sigma(\iota, (\tau e\langle\theta'\rangle)\upharpoonright_\theta)
\end{aligned}$$

The first equality above follows by the second part of Definition 107), the second by the induction hypothesis, the third by Definition 108, the fourth again by the second part of Definition 107, and the fifth by Definition 105. This concludes our proof. \square

15.8 Algorithms for Parametric Trace Monitoring

We next propose two monitoring algorithms for parametric properties. Our unoptimized but easier to understand algorithm is easily derived from the parametric trace slicing algorithm in Figure 15.2. Our second algorithm is an online optimization of the first, which significantly reduces the size of the search space for compatible parameter instances when a new event is received.

15.8.1 Unoptimized but Simpler Algorithm

Analyzing the definition of a parametric monitor (Definition 108), the first thing we note is that its state space is not only infinite, but it is not even enumerable. Therefore, a first challenge in monitoring parametric properties is how to represent the states of the parametric monitor. Inspired by the algorithm for trace slicing in Figure 15.2, we encode functions $[[X \rightarrow V] \rightarrow S]$ as tables with entries indexed by parameter instances in $[X \rightarrow V]$ and with contents states in S . Following similar arguments as in the proof of the trace slicing algorithm, such tables will have a finite number of entries provided that each event instantiates only a finite number of parameters.

Figure 15.3 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X . P$ and M a monitor for P , $\mathbb{B}\langle X \rangle(M)$ yields

Algorithm $\mathbb{B}\langle X \rangle(M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma))$
Input: finite parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$
Output: mapping $\Gamma : [[X \rightarrow V] \rightarrow \mathcal{C}]$ and set $\Theta \subseteq [X \rightarrow V]$

```

1  $\Delta \leftarrow \perp$ ;  $\Delta(\perp) \leftarrow \mathfrak{B}$ ;  $\Theta \leftarrow \{\perp\}$ 
2 foreach parametric event  $e\langle\theta\rangle$  in order in  $\tau$  do
3   : foreach  $\theta' \in \{\theta\} \sqcup \Theta$  do
4     :  $\Delta(\theta') \leftarrow \sigma(\Delta(\max(\theta')_\Theta), e)$ 
5     :  $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$  // a message may be output here
6   : endfor
7   :  $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$ 
8 endfor

```

Figure 15.3: Parametric monitoring algorithm $\mathbb{B}\langle X \rangle$

a monitor that is equivalent to $\Lambda X . M$, that is, a monitor for $\Lambda X . P$. Section 15.9 shows one way to use this algorithm: a monitor M is first synthesized from the base property P , then that monitor M is used to synthesize the monitor $\mathbb{B}\langle X \rangle(M)$ for the parametric property $\Lambda X . P$. $\mathbb{B}\langle X \rangle(M)$ follows very closely the algorithm for trace slicing in Figure 15.2, the main difference being that trace slices are processed, as generated, by M : instead of calculating the trace slice of θ' by appending base event e to the corresponding existing trace slice in line 4 of $\mathbb{A}\langle X \rangle$, we now calculate and store in table Δ the state of the *monitor instance* corresponding to θ' by sending e to the corresponding existing monitor instance (line 4 in $\mathbb{B}\langle X \rangle(M)$); at the same time we also calculate the output corresponding to that monitor instance and store it in table Γ . In other words, we replace trace slices in $\mathbb{A}\langle X \rangle$ by local monitors processing online those slices. In our implementation in Section 15.9, we also check whether $\Gamma(\theta')$ at line 5 violates the property and, if so, an error message including θ' is output to the user.

Definition 84 Given $\tau \in \mathcal{E}\langle X \rangle^*$, let $\mathbb{B}\langle X \rangle(M)(\tau).\Theta$ and $\mathbb{B}\langle X \rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X \rangle(M)(\theta).\Gamma$ be the three data-structures maintained by the algorithm $\mathbb{B}\langle X \rangle(M)$ in Figure 15.3 after processing τ . Let $\perp \mapsto \mathfrak{B} = \mathbb{B}\langle X \rangle(M)(\epsilon).\Delta \in [[X \rightarrow V] \rightarrow S]$ be the partial map taking $\perp \in [X \rightarrow V]$ to ι and undefined elsewhere.

Corollary 7 The following hold for any $\tau \in \mathcal{E}\langle X \rangle^*$:

1. $\text{Dom}(\mathbb{B}\langle X \rangle(M)(\tau).\Delta) = \mathbb{B}\langle X \rangle(M)(\tau).\Theta = \Theta_\tau$;
2. $\mathbb{B}\langle X \rangle(M)(\tau).\Delta(\theta) = \sigma(\iota, \tau \upharpoonright_\theta)$ and
 $\mathbb{B}\langle X \rangle(M)(\tau).\Gamma(\theta) = \gamma(\sigma(\iota, \tau \upharpoonright_\theta))$ for any $\theta \in \mathbb{B}\langle X \rangle(M)(\tau).\Theta$;
3. $\sigma(\iota, \tau \upharpoonright_\theta) = \mathbb{B}\langle X \rangle(M)(\tau).\Delta(\max(\theta)_{\mathbb{B}\langle X \rangle(M)(\tau).\Theta})$ and
 $\gamma(\sigma(\iota, \tau \upharpoonright_\theta)) = \mathbb{B}\langle X \rangle(M)(\tau).\Gamma(\max(\theta)_{\mathbb{B}\langle X \rangle(M)(\tau).\Theta})$ for any $\theta \in [X \rightarrow V]$.

Proof: Follows from Theorem 23 and the discussion above. \square

We next show how to associate a formal monitor to the algorithm $\mathbb{B}\langle X \rangle(M)$ in Figure 15.3:

Definition 85 For the algorithm $\mathbb{B}\langle X \rangle(M)$ in Figure 15.3, let

$$\mathcal{M}_{\mathbb{B}\langle X \rangle(M)} = (R, \mathcal{E}\langle X \rangle, [[X \rightarrow V] \rightarrow \mathcal{C}], \perp \mapsto \mathfrak{B}, \text{next}, \text{out})$$

be the monitor defined as follows:

- $R \subseteq [[X \rightarrow V] \rightarrow S]$ is the set

$$\{\mathbb{B}\langle X \rangle(M)(\tau).\Delta \mid \tau \in \mathcal{E}\langle X \rangle^*\}$$

of reachable Δ 's in $\mathbb{B}\langle X \rangle(M)$, and

- $\text{next} : R \times \mathcal{E}\langle X \rangle \rightarrow R$ and $\text{out} : R \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$ are functions defined as follows, where $\tau \in \mathcal{E}\langle X \rangle^*$, $e \in \mathcal{E}$, and $\theta \in [X \rightarrow V]$:

$$\begin{aligned} \text{next}(\mathbb{B}\langle X \rangle(M)(\tau).\Delta, e\langle \theta \rangle) &= \mathbb{B}\langle X \rangle(M)(\tau e\langle \theta \rangle).\Delta, \text{ and} \\ \text{out}(\mathbb{B}\langle X \rangle(M)(\tau).\Delta)(\theta) &= \mathbb{B}\langle X \rangle(M)(\tau).\Gamma(\max(\theta)_{\mathbb{B}\langle X \rangle(M)(\tau).\Theta}). \end{aligned}$$

Theorem 24 $\mathcal{M}_{\mathbb{B}\langle X \rangle(M)} \equiv \Lambda X . M$ for any monitor M .

Proof: All we have to do is to show that, for any $\tau \in \mathcal{E}\langle X \rangle^*$, $\text{out}(\text{next}(\perp \mapsto \mathfrak{B}, \tau))$ and $(\Lambda X . \gamma)((\Lambda X . \sigma)(\lambda\theta.\iota, \tau))$ are equal as total functions in $[[X \rightarrow V] \rightarrow \mathcal{C}]$. Let $\theta \in [X \rightarrow V]$; then:

$$\begin{aligned} \text{out}(\text{next}(\perp \mapsto \mathfrak{B}, \tau))(\theta) &= \text{out}(\mathbb{B}\langle X \rangle(M)(\tau).\Delta)(\theta) \\ &= \mathbb{B}\langle X \rangle(M)(\tau).\Gamma(\max(\theta)_{\mathbb{B}\langle X \rangle(M)(\tau).\Theta}) \\ &= \gamma(\sigma(\lambda\theta.\iota, \tau \upharpoonright_\theta)) \\ &= \gamma((\Lambda X . \sigma)(\lambda\theta.\iota, \tau)(\theta)) \\ &= (\Lambda X . \gamma)((\Lambda X . \sigma)(\lambda\theta.\iota, \tau))(\theta). \end{aligned}$$

The first equality above follows inductively by the definition of *next* (Definition 85), noticing that $\perp \mapsto \beta = \mathbb{B}\langle X \rangle(M)(\epsilon).\Delta$. The second equality follows by the definition of *out* (Definition 85) and the third by β in Corollary 7. The fourth equality above follows inductively by the definition of $\Lambda X.\sigma$ (Definition 108) and has already been proved as part of the proof of Proposition 41. Finally, the fifth equality follows by the definition of $\Lambda X.\gamma$ (Definition 108).

Therefore, $\mathcal{M}_{\mathbb{B}\langle X \rangle(M)}$ and $\Lambda X.M$ define the same property. \square

Corollary 8 *If M is a monitor for P and X is a set of parameters, then $\mathcal{M}_{\mathbb{B}\langle X \rangle(M)}$ is a monitor for parametric property $\Lambda X.P$.*

Proof: With the notation in Definition 80, Theorem 24 implies that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X \rangle(M)}} = \mathcal{P}_{\Lambda X.M}$. By Proposition 41 we have that $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$. Finally, since $P = \mathcal{P}_M$ by Proposition 36, we conclude that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X \rangle(M)}} = \Lambda X.P$. \square

15.8.2 Optimized Algorithm

Algorithm $\mathbb{C}\langle X \rangle$ in Figure 16.5 refines Algorithm $\mathbb{B}\langle X \rangle$ in Figure 15.3 for efficient online monitoring. Since no complete trace is given in online monitoring, $\mathbb{C}\langle X \rangle$ focuses on actions to carry out when a parametric event $e(\theta)$ arrives; in other words, it essentially expands the body of the outer loop in $\mathbb{B}\langle X \rangle$ (lines 3 to 7 in Figure 15.3). The direct use of $\mathbb{B}\langle X \rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires search for all parameter instances in Θ that are compatible with θ ; this search can be very expensive. $\mathbb{C}\langle X \rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates further optimizations. While $\mathbb{B}\langle X \rangle$ did not require that θ in $e(\theta)$ be of finite domain, $\mathbb{C}\langle X \rangle$ needs that requirement in order to terminate. Note that in practice $\text{Dom}(\theta)$ is always finite (because the program state is finite).

$\mathbb{C}\langle X \rangle$ uses three tables: Δ , \mathcal{U} and Γ . Δ and Γ are the same as Δ and Γ in $\mathbb{B}\langle X \rangle$, respectively. \mathcal{U} is an auxiliary data structure used to optimize the search “for all $\theta' \in \{\theta\} \sqcup \Theta$ ” in $\mathbb{B}\langle X \rangle$ (line 3 in Figure 15.3). It maps each parameter instance θ into the finite set of parameter instances encountered in Δ so far that are strictly more informative than θ , i.e., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \text{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$. Another major difference between $\mathbb{B}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ is that $\mathbb{C}\langle X \rangle$ does *not* maintain Θ during computation;

Algorithm $\mathbb{C}\langle X \rangle(M = (S, \mathcal{E}, \mathcal{C}, \iota, \sigma, \gamma))$
 Globals: mapping $\Delta : [[X \rightarrow V] \rightarrow S]$ and
 mapping $\mathcal{U} : [X \rightarrow V] \rightarrow \mathcal{P}_f([X \rightarrow V])$ and
 mapping $\Gamma : [[X \rightarrow V] \rightarrow \mathcal{C}]$
 Initialization: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightarrow V]$, $\Delta(\perp) \leftarrow \iota$

```

function main( $e\langle\theta\rangle$ )
1  if  $\Delta(\theta)$  undefined then
2  : foreach  $\theta_{max} \sqsubset \theta$  (in reversed topological order) do
3  : : if  $\Delta(\theta_{max})$  defined then
4  : : : goto 7
5  : : endif
6  : endfor
7  : defineTo( $\theta, \theta_{max}$ )
8  : foreach  $\theta_{max} \sqsubset \theta$  (in reversed topological order) do
9  : : foreach  $\theta_{comp} \in \mathcal{U}(\theta_{max})$  that is compatible with  $\theta$  do
10 : : : if  $\Delta(\theta_{comp} \sqcup \theta)$  undefined then
11 : : : : defineTo( $\theta_{comp} \sqcup \theta, \theta_{comp}$ )
12 : : : : endif
13 : : : endfor
14 : : endfor
15 : endif
16 : foreach  $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$  do
17 : :  $\Delta(\theta') \leftarrow \sigma(\Delta(\theta), e)$ 
18 : :  $\Gamma(\theta') \leftarrow \sigma(\Delta(\theta'))$ 
19 : endfor

function defineTo( $\theta, \theta'$ )
1   $\Delta(\theta) \leftarrow \Delta(\theta')$ 
2  foreach  $\theta'' \sqsubset \theta$  do
3  :  $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$ 
4  endfor

```

Figure 15.4: Online parametric monitoring algorithm $\mathbb{C}\langle X \rangle$

instead, Θ is implicitly captured by the domain of Δ in $\mathbb{C}\langle X \rangle$. Intuitively, the Θ at the beginning/end of the body of the outer loop in $\mathbb{B}\langle X \rangle$ is the $\text{Dom}(\Delta)$ at the beginning/end of $\mathbb{C}\langle X \rangle$, respectively. However, Θ is fixed during the loop at lines 3 to 6 in $\mathbb{B}\langle X \rangle$ and updated atomically in line 7, while $\text{Dom}(\Delta)$ can be changed at any time during the execution of $\mathbb{C}\langle X \rangle$.

$\mathbb{C}\langle X \rangle$ is composed of two functions, `main` and `defineTo`. The `defineTo` function takes two parameter instances, θ and θ' , and adds a new entry corresponding to θ into Δ and \mathcal{U} . Specifically, it sets $\Delta(\theta)$ to $\Delta(\theta')$ and adds θ into the set $\mathcal{U}(\theta'')$ for each $\theta'' \sqsubset \theta$.

The `main` function differentiates two cases when a new event $e\langle\theta\rangle$ is received and processed. The simpler case is that Δ is already defined on θ , i.e., $\theta \in \Theta$ at the beginning of the iteration of the outer loop in $\mathbb{B}\langle X \rangle$. In this case, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta \sqsubseteq \theta'\} \subseteq \Theta$, so the lines 3 to 6 in $\mathbb{B}\langle X \rangle$ become precisely the lines 16 to 19 in $\mathbb{C}\langle X \rangle$. In the other case, when Δ is not already defined on θ , `main` takes two steps to handle e . The first step searches for new parameter instances introduced by $\{\theta\} \sqcup \Theta$ and adds entries for them into Δ (lines 2 to 14). We first add an entry to Δ for θ at lines 2 to 7. Then we search for all parameter instances θ_{comp} that are compatible with θ , making use of \mathcal{U} (lines 8 and 9); for each such θ_{comp} , an appropriate entry is added to Δ for its lub with θ , and \mathcal{U} updated accordingly (lines 10 to 12). This way, Δ will be defined on all the new parameter instances introduced by $\{\theta\} \sqcup \Theta$ after the first step. In the second step, the related monitor states and outputs are updated in a similar way as in the first case (lines 16 to 19). It is interesting to note how $\mathbb{C}\langle X \rangle$ searches at lines 2 and 8 for the parameter instance $\max(\theta)_{\Theta}$ that $\mathbb{B}\langle X \rangle$ refers to at line 4 in Figure 15.3: it enumerates all the $\theta_{max} \sqsubset \theta$ in *reversed topological order* (from larger to smaller); 1. in Proposition 29 guarantees that the maximum exists and, since it is unique, our search will find it.

Correctness of $\mathbb{C}\langle X \rangle$. We prove the correctness of $\mathbb{C}\langle X \rangle$ by showing that it is equivalent to the body of the outer loop in $\mathbb{B}\langle X \rangle$. Suppose that parametric trace τ has already been processed by both $\mathbb{C}\langle X \rangle$ and $\mathbb{B}\langle X \rangle$, and a new event $e\langle\theta\rangle$ is to be processed next.

Let us first note that $\mathbb{C}\langle X \rangle$ terminates if $\text{Dom}(\theta)$ is finite. Indeed, if $\text{Dom}(\theta)$ is finite then there is only a finite number of partial maps less informative than θ , that is, only a finite number of iterations for the loops at lines 2 and 8 in `main`; since \mathcal{U} is only updated at line 3 in `defineTo`, $\mathcal{U}(\theta)$ is finite for any $\theta \in [X \rightarrow V]$ and thus the loop at line 9 in `main` also terminates. Assuming that running the base monitor M takes constant time, the worst

case complexity of $\mathbb{C}\langle X \rangle(M)$ is $O(k \times l)$ to process $e\langle \theta \rangle$, where k is $2^{|\text{Dom}(\theta)|}$ and l is the number of incompatible parameter instances in τ . Parametric properties often have a fixed and small number of parameters, in which case k is not significant. Depending on the trace, l can unavoidably grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

Lemma 1 *In the algorithm $\mathbb{C}\langle X \rangle$ in Figure 16.5, $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \text{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ before and after each execution of `defineTo`, for all $\theta \in [X \rightarrow V]$.*

Proof: By how $\mathbb{C}\langle X \rangle$ is initialized, for any $\theta \in [X \rightarrow V]$ we have $\emptyset = \mathcal{U}(\theta) = \{\theta' \mid \theta' \in \text{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ before the first execution of `defineTo`. Now suppose that $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \text{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ for any $\theta \in [X \rightarrow V]$ before an execution of `defineTo` and show that it also holds after the execution of `defineTo`. Since `defineTo`(θ, θ') adds a new parameter instance θ into $\text{Dom}(\Delta)$ and also adds θ into the set $\mathcal{U}(\theta'')$ for any $\theta'' \in [X \rightarrow V]$ with $\theta'' \sqsubset \theta$, we still have $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in \text{Dom}(\Delta) \text{ and } \theta \sqsubset \theta'\}$ for any $\theta \in [X \rightarrow V]$ after the execution of `defineTo`. Also, the only way $\mathbb{C}\langle X \rangle$ can add a new parameter instance θ into $\text{Dom}(\Delta)$ is by using `defineTo`. Therefore the lemma holds. \square

The next theorem proves the correctness of $\mathbb{C}\langle X \rangle$. Before we state and prove it, let us recall some previously introduced notation and also introduce some new useful notation. First, recall from Definition 84 that $\mathbb{B}\langle X \rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X \rangle(M)(\tau).\Gamma$ are the Δ and Γ data-structures of $\mathbb{B}\langle X \rangle(M)$ after it processes trace τ . Also, recall that we fixed parametric trace τ and event $e\langle \theta \rangle$. For clarity, let $\mathcal{U}_{\mathbb{C}}$, $\Delta_{\mathbb{C}}$, and $\Gamma_{\mathbb{C}}$ be the three data-structures maintained by $\mathbb{C}\langle X \rangle(M)$ (in other words, we index the data-structures with the symbol \mathbb{C}). Let $\Delta_{\mathbb{C}}^b$ and $\Gamma_{\mathbb{C}}^b$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when `main`($e\langle \theta \rangle$) begins (“ b ” stays for “at the beginning”); let $\Delta_{\mathbb{C}}^e$ and $\Gamma_{\mathbb{C}}^e$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when `main`($e\langle \theta \rangle$) ends (“ e ” stays for “at the end”; and let $\Delta_{\mathbb{C}}^m$ and $\mathcal{U}_{\mathbb{C}}^m$ be the $\Delta_{\mathbb{C}}$ and $\mathcal{U}_{\mathbb{C}}$ when `main`($e\langle \theta \rangle$) reaches line 16 (“ m ” stays for “in the middle”).

Theorem 25 *The following hold:*

1. $\text{Dom}(\Delta_{\mathbb{C}}^m) = \{\perp, \theta\} \sqcup \text{Dom}(\Delta_{\mathbb{C}}^b)$;
2. $\Delta_{\mathbb{C}}^m(\theta') = \Delta_{\mathbb{C}}^m(\max(\theta')_{\text{Dom}(\Delta_{\mathbb{C}}^b)})$, for all $\theta' \in \text{Dom}(\Delta_{\mathbb{C}}^m)$;
3. If $\Delta_{\mathbb{C}}^b = \mathbb{B}\langle X \rangle(M)(\tau).\Delta$ and $\Gamma_{\mathbb{C}}^b = \mathbb{B}\langle X \rangle(M)(\tau).\Gamma$, then $\Delta_{\mathbb{C}}^e = \mathbb{B}\langle X \rangle(M)(\tau e\langle \theta \rangle).\Delta$ and $\Gamma_{\mathbb{C}}^e = \mathbb{B}\langle X \rangle(M)(\tau e\langle \theta \rangle).\Gamma$.

Proof: Let $\Theta_{\mathbb{C}} = \text{Dom}(\Delta_{\mathbb{C}}^b) = \text{Dom}(\Delta_{\mathbb{B}}(\tau))$ and $\Delta_{\mathbb{B}}(\tau) = \mathbb{B}\langle X \rangle(M)(\tau\langle\theta\rangle).\Delta$ for simplicity.

1. There are two cases to analyze, depending upon whether θ is in $\Theta_{\mathbb{C}}$ or not. If $\theta \in \Theta_{\mathbb{C}}$ then the lines 2 to 14 are skipped and $\text{Dom}(\Delta_{\mathbb{C}})$ remains unchanged, that is, $\{\perp, \theta\} \sqcup \Theta_{\mathbb{C}} = \Theta_{\mathbb{C}} = \text{Dom}(\Delta_{\mathbb{C}}^b) = \text{Dom}(\Delta_{\mathbb{C}}^m)$ when $\text{main}(e\langle\theta\rangle)$ reaches line 16. If $\theta \notin \Theta_{\mathbb{C}}$ then lines 2 to 14 are executed to add new parameter instances into $\text{Dom}(\Delta_{\mathbb{C}})$. First, an entry for θ will be added to $\Delta_{\mathbb{C}}$ at line 7. Second, an entry for $\theta_{\text{comp}} \sqcup \theta$ will be added to $\Delta_{\mathbb{C}}$ at line 11 (if $\Delta_{\mathbb{C}}$ not already defined on $\theta_{\text{comp}} \sqcup \theta$) eventually for any $\theta_{\text{comp}} \in \Theta_{\mathbb{C}}$ compatible with θ : that is because θ_{max} can also be \perp at line 8, in which case Lemma 1 implies that $\mathcal{U}(\theta_{\text{max}}) = \Theta_{\mathbb{C}}$. Therefore, when line 16 is reached, $\text{Dom}(\Delta_{\mathbb{C}}^m)$ is defined on all the parameter instances in $\{\theta\} \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}})$. Since $\perp \in \Theta_{\mathbb{C}}$, the latter equals $\{\theta\} \sqcup \Theta_{\mathbb{C}}$, and since $\Delta_{\mathbb{C}}^m$ remains defined on $\Theta_{\mathbb{C}}$, we conclude that $\Delta_{\mathbb{C}}^m$ is defined on all instances in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}}$, which by 5. and 7. in Proposition 25 equals $\{\perp, \theta\} \sqcup \Theta_{\mathbb{C}}$.

2. We analyze the same two cases as above. If $\theta \in \Theta_{\mathbb{C}}$ then lines 2 to 14 are skipped and $\text{Dom}(\Delta_{\mathbb{C}})$ remains unchanged. Then $\max(\theta')_{\Theta_{\mathbb{C}}} = \theta'$ for each $\theta' \in \text{Dom}(\Delta_{\mathbb{C}}^m)$, so the result follows. Suppose now that $\theta \notin \Theta_{\mathbb{C}}$. By 1. and its proof, each $\theta' \in \text{Dom}(\Delta_{\mathbb{C}}^m)$ is either in $\Theta_{\mathbb{C}}$ or otherwise in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$. The result immediately holds when $\theta' \in \Theta_{\mathbb{C}}$ as $\max(\theta')_{\Theta_{\mathbb{C}}} = \theta'$ and $\Delta(\theta')$ stays unchanged until line 16. If $\theta' \in (\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$ then $\Delta(\theta')$ is set at either line 7 ($\theta' = \theta$) or at line 11 ($\theta' \neq \theta$):

(a) For line 7, the loop at lines 2 to 6 checks all the parameter instances that are less informative than θ to find the first one in $\Theta_{\mathbb{C}}$ in reversed topological order (i.e., if $\theta_1 \sqsubset \theta_2$ then θ_2 will be checked before θ_1). Since by 1. in Proposition 29 we know that $\max(\theta)_{\Theta_{\mathbb{C}}} \in \Theta_{\mathbb{C}}$ exists (and it is unique), the loop at lines 2 to 6 will break precisely when $\theta_{\text{max}} = \max(\theta)_{\Theta_{\mathbb{C}}}$, so the result holds when $\theta' = \theta$ because of the entry introduced for θ in $\Delta_{\mathbb{C}}$ at line 7 and because the remaining lines 8 to 14 do not change $\Delta_{\mathbb{C}}(\theta)$.

(b) When $\Delta_{\mathbb{C}}(\theta')$ is set at line 11, note that the loop at lines 8 to 14 also iterates over all $\theta_{\text{max}} \sqsubset \theta$ in reversed topological order, so $\theta' = \theta_{\text{comp}} \sqcup \theta$ for some $\theta_{\text{comp}} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta_{\text{max}} \sqsubset \theta_{\text{comp}}$, where $\theta_{\text{max}} \sqsubset \theta$ is such that there is no other θ'_{max} with $\theta_{\text{max}} \sqsubset \theta'_{\text{max}} \sqsubset \theta$ and $\theta' = \theta'_{\text{comp}} \sqcup \theta$ for some $\theta'_{\text{comp}} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta'_{\text{max}} \sqsubset \theta'_{\text{comp}}$. We claim that there is only one such θ_{comp} , which is precisely $\max(\theta')_{\Theta_{\mathbb{C}}}$: Let θ'_{comp} be the parameter instance $\max(\theta')_{\Theta_{\mathbb{C}}}$. The above implies that $\theta_{\text{comp}} \sqsubseteq \theta'_{\text{comp}} \sqsubseteq \theta'$. Also, $\theta'_{\text{comp}} \sqcup \theta = \theta'$ because $\theta' = \theta_{\text{comp}} \sqcup \theta \sqsubseteq \theta'_{\text{comp}} \sqcup \theta \sqsubseteq \theta'$. Let θ'_{max} be $\theta'_{\text{comp}} \sqcap \theta$, that is, the largest with $\theta'_{\text{max}} \sqsubseteq \theta'_{\text{comp}}$ and $\theta'_{\text{max}} \sqsubseteq \theta$ (we let its

existence as exercise). It is relatively easy to see now that $\theta_{comp} \sqsubset \theta'_{comp}$ implies $\theta_{max} \sqsubset \theta'_{max}$ (we let it as an exercise, too), which contradicts the assumption of this case that $\Delta_{\mathbb{C}}$ was not defined on θ' . Therefore, $\theta_{comp} = \max(\theta']_{\Theta_{\mathbb{C}}}$ before line 11 is executed, which means that, after line 11 is executed, $\Delta_{\mathbb{C}}(\theta') = \Delta_{\mathbb{C}}(\max(\theta']_{\Theta_{\mathbb{C}}})$; moreover, none of these will be changed anymore until line 16 is reached, which proves our result.

3. Since Γ is updated according to Δ in both $\mathbb{C}\langle X \rangle$ and $\mathbb{B}\langle X \rangle$, it is enough to prove that $\Delta_{\mathbb{C}}^e = \Delta_{\mathbb{B}}(\tau e)$. For $\mathbb{B}\langle X \rangle$, we have

- 1) $\text{Dom}(\Delta_{\mathbb{B}}(\tau e)) = \{\perp, \theta\} \sqcup \Theta_{\mathbb{C}} = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}};$
- 2) $\forall \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}, \Delta_{\mathbb{B}}(\tau e)(\theta') = \sigma(\Delta_{\mathbb{B}}(\tau)(\max(\theta']_{\Theta_{\mathbb{C}}}), e);$
- 3) $\forall \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}, \Delta_{\mathbb{B}}(\tau e)(\theta') = \Delta_{\mathbb{B}}(\tau)(\theta').$

So we only need to prove that

- 1) $\text{Dom}(\Delta_{\mathbb{C}}^e) = \{\perp, \theta\} \sqcup \Theta_{\mathbb{C}};$
- 2) $\forall \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}, \Delta_{\mathbb{C}}^e(\theta') = \sigma(\Delta_{\mathbb{C}}^b(\max(\theta']_{\Theta_{\mathbb{C}}}), e);$
- 3) $\forall \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}, \Delta_{\mathbb{C}}^e(\theta') = \Delta_{\mathbb{C}}^b(\theta').$

By 1., we have $\text{Dom}(\Delta_{\mathbb{C}}^m) = \{\perp, \theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\text{Dom}(\Delta_{\mathbb{C}})$, $\text{Dom}(\Delta_{\mathbb{C}}^e) = \text{Dom}(\Delta_{\mathbb{C}}^m) = \{\perp, \theta\} \sqcup \Theta_{\mathbb{C}}$. 1) holds.

By 2. and Lemma 1, $\Delta_{\mathbb{C}}^m(\theta') = \Delta_{\mathbb{C}}^b(\max(\theta']_{\Theta_{\mathbb{C}}})$ for any $\theta' \in \text{Dom}(\Delta_{\mathbb{C}}^m)$. Also, notice that line 17 sets $\Delta_{\mathbb{C}}(\theta')$ to $\sigma(\Delta_{\mathbb{C}}(\theta'), e)$, which is $\sigma(\Delta_{\mathbb{C}}^b(\max(\theta']_{\Theta_{\mathbb{C}}}), e)$, for the θ' in the loop. So, to show 2) and 3), we only need to prove that the loop at line 16 to 19 iterates over $\{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\mathcal{U}_{\mathbb{C}}$, we need to show $\{\theta\} \sqcup \mathcal{U}_{\mathbb{C}}^m(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since $\text{Dom}(\Delta_{\mathbb{C}}^m) = \{\perp, \theta\} \sqcup \Theta_{\mathbb{C}}$, we have $\{\theta\} \sqcup \text{Dom}(\Delta_{\mathbb{C}}^m) = \{\theta\} \sqcup (\{\perp, \theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup ((\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}})$. By Proposition 25, $\{\theta\} \sqcup \text{Dom}(\Delta_{\mathbb{C}}^m) = (\{\theta\} \sqcup (\{\theta\} \sqcup \Theta_{\mathbb{C}})) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Also, as $\theta \in \text{Dom}(\Delta_{\mathbb{C}}^m)$, we have $\{\theta\} \sqcup \text{Dom}(\Delta_{\mathbb{C}}^m) = \{\theta' \mid \theta' \in \text{Dom}(\Delta_{\mathbb{C}}^m) \text{ and } \theta \sqsubseteq \theta'\} = \{\theta\} \sqcup \mathcal{U}_{\mathbb{C}}^m(\theta)$ by Lemma 1. So $\{\theta\} \sqcup \mathcal{U}_{\mathbb{C}}^m(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. \square

We conclude this section with a discussion on the complexity of the parametric monitoring algorithms $\mathbb{A}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ above. Note that, in the worst case, to process a newly received parametric event $e\langle \theta \rangle$ after $\mathbb{A}\langle X \rangle$ or $\mathbb{C}\langle X \rangle$ has already processed a parametric trace τ , each of $\mathbb{A}\langle X \rangle$ or $\mathbb{C}\langle X \rangle$ takes at least linear time/space in the number of θ -compatible parameter instances occurring in events in τ . Indeed, $\mathbb{A}\langle X \rangle$ iterates explicitly through all such parameter instances (line 3 in Figure 15.3), while $\mathbb{C}\langle X \rangle$ optimizes this traversal by only enumerating through maximal parameter instances; in the worst case, we can assume that τ is such that each event comes with a new parameter instance which is maximal, so in the worst case $\mathbb{A}\langle X \rangle$ and $\mathbb{C}\langle X \rangle$ can take linear time/space in τ to process $e\langle \theta \rangle$, which is, nevertheless,

bad. Indeed, it means that monitoring some traces is incrementally slower (with no upper bound) as events are received, until the monitor eventually runs out of resources.

Unfortunately, there is nothing to be fundamentally done to avoid the problem above. It is an inherent problem of parametric monitoring. Consider, for example, the “authenticate before use” parametric property specified in Section 15.2.2 using parametric LTL as $\Lambda k. \Box(\text{use}\langle k \rangle \rightarrow \Diamond \text{authenticate}\langle k \rangle)$; to make it clear that events depend on the parameter key k , we tagged them with the key. Without any knowledge about the semantics of the program to monitor, any monitor for this property *must* store all the authenticated keys, i.e., all the instances of the parameter k . Indeed, without that, there is no way to know whether a key instance has been authenticated or not when a `use` event is observed on that key. The number of such key instances is theoretically unbounded, so, in the worst case, any monitor for this property can be incrementally slower and eventually run out of resources.

As seen in Section 15.9, the runtime overhead due to opaque monitoring of parametric properties tends to be manageable in practice. By “opaque” we mean that no semantic information about the source code of the monitored program is used. If the lack of an efficiency guarantee is a problem in some applications, then the alternative is to statically analyze the monitored program and to use the obtained semantic information to eliminate the need for monitoring. For example, static analyses like those in [192, 40, 42, 92] may significantly reduce the need for instrumentation, even eliminate it completely. Moreover, model-checking techniques for parametric properties could also be used for actually proving that the properties hold and thus they need not be monitored; however, we are not aware of model checking approaches to verifying parametric properties as presented in this paper.

15.9 Implementation in JavaMOP and RV

The discussed parametric monitoring technique is now fully implemented in two runtime verification systems, namely in JavaMOP (see <http://javamop.org>) and in RV [197] (developed by a startup company, Runtime Verification, Inc.; the RV system is currently publicly unavailable – contact the first author for an NDA-protected version of RV). Here we first informally discuss several optimizations implemented in the two runtime verification systems, and then we discuss our experiments and the evaluation of the two systems.

15.9.1 Implementation Optimizations

Both JavaMOP and RV apply several optimizations to the algorithm $\mathbb{C}\langle X \rangle$ in Section 15.8.2, to reduce its runtime overhead. These are not discussed in depth here, because they are orthogonal to the main objective of this paper.

Optimizations in JavaMOP

Note that $\mathbb{C}\langle X \rangle$ iterates through all the possible parameter instances that are less informative than θ in three different loops: at lines 2 and 8 in the main function, and at line 2 in the `defineTo` function. Hence, it is important to reduce the number of such instances in each loop. Even though our semantics and theoretical algorithms for parametric monitoring in this paper work with infinite sets of parameters, our current implementation in JavaMOP assumes that the set of parameters X is bounded and fixed apriori (declared as part of the specification to monitor). A simple analysis of the events appearing in the specification allows to quickly detect parameter instances that can never appear as lubs of instances of parameters carried by events; maintaining any space for those in Δ , or Γ , or iterating over them in the above mentioned loops, is a waste. For example, if a specification contains only two event definitions, $e_1\langle p_1 \rangle$ and $e_2\langle p_1, p_2 \rangle$, parameter instances defining only parameter p_2 can never appear as lubs of observed parameter instances. A static analysis of the specification, discussed in [57, 40], exhaustively explores all possible event combinations that can lead to situations of interest to the property, such as to violation, validation, etc. Such information is useful to reduce the number of loop iterations by skipping iterations over parameter instances that cannot affect the result of monitoring. These static analyses are currently used at compile time in our new JavaMOP implementation to unroll the loops in $\mathbb{C}\langle X \rangle$ and reduce the size of Δ and \mathcal{U} .

Another optimization is based on the observation that it is convenient to start the monitoring process only when certain events are received. Such events are called monitor creation events in [62]. The parameter instances carried by such creation events may also be used to reduce the number of parameter instances that need to be considered. An extreme, yet surprisingly common case is when creation events instantiate *all* the property parameters. In this case, the monitoring process does not need to search for compatible parameter instances even when an event with an incomplete parameter instance is observed. The old JavaMOP [62] supported only traces whose monitoring started with a fully instantiated monitor creation event;

this was perceived (and admitted) as a performance-tradeoff limitation of JavaMOP [24] (and [62]). Interestingly, it now becomes just a common-case optimization of our novel, general and unrestricted technique presented here.

Optimizations in RV

The RV system implements all the optimizations in JavaMOP and adds two other important optimizations that significantly reduce the overhead.

The first additional optimization of RV is a non-trivial garbage-collector [160]. Note that JavaMOP also has a garbage collector, but it only does the obvious: it garbage collects a monitor instance only when all its corresponding parameter instances are collected. RV performs a static analysis of the property to monitor and, based on that, it garbage collects a monitor instance as soon as it realizes that it can never trigger in the future. This can happen when any triggering behavior needs at least one event that can only be generated in the presence of a parameter instance that is already dead. Consider, for example, the safe iterator example in Section 15.2.3, and consider that iterator i_7 is created for collection c_3 . Then a monitor instance corresponding to the parameter instance $\langle c_3 \ i_7 \rangle$ is created and manged. Suppose that, at some moment, the iterator i_7 is garbage collected by the JVM. Can the monitor instance corresponding to $\langle c_3 \ i_7 \rangle$ be garbage collected? Not in JavaMOP, because, for safety, JavaMOP collects a monitor only when all its parameter instances are collected, and in this case c_3 is still alive. However, this monitor is flagged for garbage collection in RV. The rationale for doing so is that the only way for the monitor to trigger is to eventually encounter a next event with i_7 as parameter, but that event can never be generated because i_7 is dead. Note, on the other hand, that the monitor $\langle c_3 \ i_7 \rangle$ cannot be garbage collected if c_3 is collected but i_7 is still alive, because the iterator alone can still violate the safe-iterator property, even if its corresponding collection is already dead.

Runtime verification systems like JavaMOP and Tracematches use off-the-shelf weak reference libraries to implement their garbage collectors. However, it turns out that these libraries, in order to be general and thus serve their purpose, perform many checks that are unnecessary in the context of monitoring. The second optimization of RV in addition to those of JavaMOP consists of a collection of data structures based on weak references, which was carefully engineered to take full advantage of the particularities of monitoring parametric properties. These data structures allow for effective indexing and lazy collection of monitors, to minimize the number of expensive traversals

of the entire pool of monitors. Moreover, RV caches monitor instances to save time when the same monitor instances are accessed frequently. For example, there is a high chance that the same iterator is accessed several times consecutively by a program, in which case saving and then retrieving the same corresponding monitor instances from the data-structures at each iterator access can take considerable unnecessary overhead.

15.9.2 Experiments and Evaluation

We next discuss our experience with using the two runtime verification systems that implement optimized variants of the parametric property monitoring techniques describe in this paper. Also, we compare their performance with that of Tracematches, which is, at our knowledge, the most efficient runtime verification system besides JavaMOP and RV. Recall that Tracematches achieves virtually the same semantics of parametric monitoring like ours, but using a considerably different approach.

Experimental Settings

We used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [36], the most up-to-date version. We also present some of the results of our experiments using the previous version of DaCapo, 2006-10 MR2 (DaCapo 2006-10), namely those for the bloat and jython benchmarks. DaCapo 9.12 does not provide the bloat benchmark from the DaCapo 2006-10, which we favor because it generates large overheads when monitoring iterator-based properties. The bloat benchmark with the UNSAFEITER specification causes 19194% runtime overhead (i.e., 192 times slower) and uses 7.7MB of heap memory in Tracematches, and causes 569% runtime overhead and uses 147MB in JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error that we were not able to understand or fix. Thus, we present the result of jython from the DaCapo 2006-10. The default data input for DaCapo was used and the -converge option to obtain the numbers after convergence within $\pm 3\%$. Instrumentation introduces a different garbage collection behavior in the monitored program, sometimes causing the program to slightly outperform the original program; this accounts for the negative overheads seen in both runtime and memory.

We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ

compiler (ajc) version 1.6.4 is used for weaving the aspects generated by JavaMOP and RV into the target benchmarks. Another AspectJ compiler, abc [23] 1.3.0, is used for weaving Tracematches properties because Tracematches is part of abc and does not work with ajc. For JavaMOP, we used the most recent release version, 2.1.2. For Tracematches, we used the most recent release version, 1.3.0, from [37], which is included in the abc compiler as an extension. To figure out the reason that some examples do not terminate when using Tracematches, we also used the abc compiler for weaving aspects generated by JavaMOP and RV. Note that JavaMOP and RV are AspectJ compiler-independent. They show similar overheads and terminate on all examples when using the abc compiler for weaving as when ajc is used. Because the overheads are similar, we do not present the results of using abc to weave JavaMOP and RV generated aspects in this paper. However, using abc to weave JavaMOP and RV properties confirms that the high overhead and non-termination come from Tracematches itself, not from the abc compiler.

The following properties are used in our experiments. Some of them were already discussed in Sections 15.1.1 and 15.2, others are borrowed from [40, 41, 195, 57].

- HASNEXT: Do not use the next element in an iterator without checking for the existence of it;
- UNSAFEITER: Do not update a collection when using the iterator interface to iterate its elements;
- UNSAFEMAPITER: Do not update a map when using the iterator interface to iterate its values or its keys;
- UNSAFESYNCCOLL: If a collection is synchronized, then its iterator also should be accessed synchronously;
- UNSAFESYNCPMAP: If a collection is synchronized, then its iterators on values and keys also should be accessed synchronized.

All of them are tested on Tracematches, JavaMOP, and RV for comparison. We also monitored all five properties at the same time in RV, which was not possible in other monitoring systems for performance reasons or structural limitations.

	ORIG (sec)	HasNext			UnsafeIter			UnsafeMapIter			UnsafeSyncColl			UnsafeSyncMap			ALL
(A)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	3.6	2119	448	116	19194	569	251	∞	1203	178	1359	746	212	1942	716	130	982
jython	8.9	13	0	0	11	0	1	150	18	3	11	1	1	10	0	0	4
avrora	13.6	45	54	55	637	311	118	∞	113	42	75	144	80	54	74	16	275
batik	3.5	3	2	3	355	9	8	∞	8	5	208	9	9	5	3	0	28
eclipse	79.0	-2	4	-1	0	-1	-1	5	-3	0	-4	2	1	∞	-1	-1	0
fop	2.0	200	49	48	350	21	13	∞	58	14	∞	78	25	∞	71	19	133
h2	18.7	89	17	13	128	9	4	1350	21	6	868	21	4	83	20	5	23
luindex	2.9	0	0	1	0	0	1	1	4	1	1	1	1	2	0	0	1
lusearch	25.3	-1	1	0	1	2	2	2	2	0	4	0	1	3	1	1	3
pmd	8.3	176	84	59	1423	162	123	∞	571	188	1818	192	76	∞	144	26	620
sunflow	32.7	47	5	3	7	2	0	9	4	1	13	6	5	17	6	6	6
tomcat	13.8	8	1	1	37	1	1	3	1	1	2	0	1	2	1	3	1
tradebeans	45.5	0	-1	1	1	1	2	5	3	-1	-1	1	2	3	1	5	2
tradesoap	94.4	1	3	0	2	1	1	2	0	1	0	0	1	2	2	5	1
xalan	20.3	4	2	2	27	7	2	10	5	2	3	2	3	4	4	3	4
(B)		TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	RV
bloat	4.9	56.8	19.3	13.2	7.7	146.8	79.0	∞	173.4	56.1	6.8	127.9	48.3	6.9	55.4	12.7	340.9
jython	5.3	5.7	4.6	4.8	4.9	4.6	4.8	6.0	19.5	4.7	5.3	4.5	4.4	5.9	4.8	5.1	4.7
avrora	4.7	4.6	12.4	9.1	4.4	136.2	15.8	∞	14.7	8.5	4.3	28.0	12.6	4.4	13.0	4.9	22.3
batik	79.1	79.2	78.7	79.3	75.2	93.6	86.6	∞	91.2	79.6	78.2	93.2	85.1	79.9	86.9	76.7	104.3
eclipse	95.9	100.8	107.6	97.1	98.3	100.0	110.3	106.9	93.8	101.1	100.4	109.2	90.1	∞	98.6	98.7	98.9
fop	20.7	97.4	47.1	52.5	24.3	24.2	29.4	∞	69.2	28.1	∞	54.8	24.8	∞	55.9	25.2	47.5
h2	265.0	267.8	598.5	565.2	267.2	266.2	262.4	312.4	688.3	268.2	271.4	690.3	265.5	271.0	718.3	270.0	283.7
luindex	6.8	5.6	5.5	5.6	6.3	6.9	6.8	7.4	8.2	6.9	7.4	7.4	7.5	7.1	7.4	11.0	11.8
lusearch	4.6	4.7	4.4	4.8	4.6	4.8	4.2	4.0	4.3	4.8	4.5	4.5	4.6	4.6	4.8	4.7	4.7
pmd	18.0	56.9	59.8	48.5	17.2	146.3	86.4	∞	212.7	93.6	20.3	238.4	84.6	∞	117.1	32.9	420.0
sunflow	4.4	4.5	4.8	4.9	4.8	4.3	4.7	4.7	4.4	4.4	5.1	4.3	4.9	4.5	4.7	4.5	4.6
tomcat	11.6	11.4	12.3	11.4	12.5	11.0	11.5	11.9	11.4	11.0	11.3	11.3	11.3	11.4	11.4	11.8	11.8
tradebeans	63.2	62.9	62.7	62.1	63.7	63.9	64.1	63.3	62.5	62.7	63.2	62.8	62.0	64.0	62.8	64.0	62.5
tradesoap	64.1	61.8	62.3	63.3	63.4	63.1	64.4	64.1	63.5	62.0	60.7	65.0	65.9	65.5	64.5	65.6	64.5
xalan	4.9	4.9	5.0	5.1	4.9	4.9	4.9	4.9	4.5	4.9	5.0	4.8	5.0	5.1	4.9	4.9	5.0

Figure 15.5: Comparison of Tracematches (TM), JavaMOP (MOP), and RV: (A) average *percent* runtime overhead; (B) total peak memory usage in MB. (convergence within 3%, ∞: not terminated after 1 hour)

Results and Discussions

Figures 17.9 and 17.10 summarize the results of the evaluation. Note that the structure of the DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for DaCapo 2006-10. Therefore, fop and pmd show higher overheads than the benchmarks using DaCapo 2006-10 from [57]. While other benchmarks show overheads less than 80% in JavaMOP, bloat, avrora, and pmd show prohibitive overhead in both runtime and memory performance. This is because they generate many iterators and all properties in this evaluation are intended to monitor iterators. For example, bloat creates

	HasNext				UNSAFEITER				UNSAFEMAPITER				UNSAFESYNCCOLL				UNSAFESYNCPMAP			
	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM
bloat	155M	1.9M	1.9M	1.8M	81M	1.9M	1.8M	1.6M	74M	3.6M	43K	3.4M	143M	4.1M	0	3.7M	161M	3.4M	0	3.4M
jython	106	50	47	26	179K	50	38	38	179K	101K	94	101K	156	100	0	83	256	150	0	122
avro	1.5M	909K	850K	765K	1.4M	909K	860K	808K	1.3M	1.2M	18	1.2M	2.4M	1.8M	0	1.7M	1.5M	909K	0	904K
batik	49K	24K	21K	21K	125K	24K	21K	10K	55K	33K	140	27K	73K	50K	0	34K	50K	26K	0	26K
eclipse	226K	7.6K	5.3K	2.9K	119K	6.6K	5.1K	2.6K	113K	22K	2.2K	7.8K	233K	15K	0	7.5K	241K	18K	0	9.2K
fop	1.0M	184K	74K	151K	709K	7.7K	7.2K	1.8K	499K	177K	67	160K	1.2M	239K	0	217K	1.2M	231K	0	213K
h2	27M	6.5M	6.0M	5.6M	12M	3.7K	3.3K	1.3K	12M	6.6M	9	6.5M	27M	6.5M	0	6.5M	27M	6.5M	0	6.5M
luindex	371	66	40	2	4.4K	65	39	0	378	183	2	59	436	132	0	30	472	125	0	25
lusearch	1.4K	131	196	114	748K	130	210	18	20K	944	338	1.4K	1.7K	262	0	402	1.8K	263	0	158
pmd	8.3M	789K	694K	571K	6.4M	551K	473K	382K	4.3M	1.3M	110K	1.1M	8.8M	1.5M	0	1.3M	8.6M	1.1M	0	999K
sunflow	2.7M	101K	101K	100K	1.3M	2	0	0	1.3M	83K	0	83K	2.7M	101K	0	101K	2.7M	101K	0	101K
tomcat	25	6	0	0	132	4	0	0	68	26	0	0	29	10	0	0	33	12	0	0
tradebeans	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
tradesoap	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0	15	6	0	0
xalan	11	3	0	0	8.9K	2	0	0	119K	20K	0	20K	13	5	0	0	15	6	0	0

Figure 15.6: Monitoring statistics: number of events (E), number of created monitors (M), number of flagged monitors (FM), number of collected monitors (CM).

1,625,770 collections and 941,466 iterators in total while 19,605 iterators coexist at the same time at peak, in an execution. avro and pmd also create many collections and iterators. Also, they call `hasNext()` 78,451,585 times, 1,158,152 times and 4,670,555 times and `next()` 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, we mainly discuss those three examples in this section, although RV shows improvements for other examples as well.

Figure 17.9 (A) shows the percent runtime overhead of Tracematches, JavaMOP, and RV. Overall, RV averages two times less runtime overhead than JavaMOP and orders of magnitude less runtime overhead than Tracematches (recall that these are the most optimized runtime verification systems). With bloat, RV shows less than 260% runtime overhead for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350% for completed runs and *crashed* for UNSAFEMAPITER. With avro, on average, RV shows 62% runtime overhead, while JavaMOP shows 139% runtime overhead and Tracematches shows 203% and hangs for UNSAFEMAPITER. With pmd, on average, RV shows 94% runtime overhead, while JavaMOP shows 231% runtime overhead and Tracematches shows 1139% and hangs for UNSAFEMAPITER and UNSAFESYNCPMAP.

Also, RV was tested with all five properties together and showed 982%,

275%, and 620% overhead, respectively, which are still faster or comparable to monitoring one of many properties alone in JavaMOP or Tracematches. The overhead for monitoring all the properties simultaneously can be slightly larger than the sum of their individual overheads since the additional memory pressure makes the JVM's garbage collection behave differently.

Figure 17.9 (B) shows the peak memory usage of the three systems. RV has lower peak memory usage than JavaMOP in most cases. The cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [24], which cannot be implemented in formalism-independent systems like RV and JavaMOP. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring bloat and pmd. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, the monitor garbage collection optimization in RV achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 17.10 shows the number of triggered events, of created monitors, of monitors flagged as unnecessary by RV's optimization, and of monitors collected by the JVM. Among the DaCapo examples, bloat, avrora, h2, pmd and sunflow generated a very large number of events (millions) in all properties, resulting in millions of monitors created in most cases. h2 does not exhibit large overhead because monitor instances in h2 have shorter lifetimes, therefore the created monitor instances are not used heavily like in bloat. sunflow has millions of events but does not create as many monitor instances as other benchmarks. When monitoring the HASNEXT and UNSAFEITER properties, RV's garbage collector effectively flagged monitors as unnecessary and most were collected by the JVM.

The experimental evaluation in this section shows that the approach to parametric trace slicing and monitoring discussed in this paper is indeed feasible, provided that it is not implemented naively. Indeed, as seen in the tables in this section, implementation optimizations make a huge difference in the runtime and memory overhead. This paper was not dedicated to optimizations and implementations; its objective was to only introduce the mathematical notions, notations, proofs and abstract algorithms underlying the semantical foundation of parametric properties and their monitoring.

Current and future implementations are and will build on this foundation, applying specific optimizations and heuristics to reduce the runtime or the memory overhead caused by monitoring.

15.10 Concluding Remarks, Future Work and Acknowledgments

A semantic foundation for parametric traces, properties and monitoring was proposed. A parametric trace slicing technique, which was discussed and proved correct, allows the extraction of all the non-parametric trace slices from a parametric slice by traversing the original trace only once and dispatching each parametric event to its corresponding slices. It thus enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. A parametric monitoring technique, also discussed and proved correct, makes use of it to monitor arbitrary parametric properties against parametric execution traces using and indexing ordinary monitors for the base, non-parametric property. Optimized implementations of the discussed techniques in JavaMOP and RV reveal that their generality, compared to the existing similar but ad hoc and limited techniques in current use, does not come at a performance expense. Moreover, further static analysis optimizations like those in [192, 40, 42, 92] may significantly reduce the runtime and memory overheads of monitoring parametric properties based on the techniques and algorithms discussed in this paper.

The parametric trace slicing technique in Section 15.6 enables the leveraging of any non-parametric, i.e., conventional, trace analysis techniques to the parametric case. We have only considered monitoring in this paper. Another interesting and potentially rewarding use of our technique could be in the context of property mining. For example, one could run the trace slicing algorithm on large benchmarks making intensive use of library classes, and then, on the obtained trace slices corresponding to particular classes or groups of classes of interest, run property mining algorithms. The mined properties, or the lack thereof, may provide insightful formal documentation for libraries, or even detect errors. Preliminary steps in this direction are reported in [183].

Acknowledgments

We would like to warmly thank the other members of the MOP team who contributed to the implementation of the new and old JavaMOP system, as well as of extensions of it, namely to Dennis Griffith, Dongyun Jin, Choonghwan Lee and Patrick Meredith. We are also grateful to Klaus Havelund, who found several errors in our new JavaMOP implementation while using it in teaching a course at Caltech, and who recommended us several simplifications in its user interface. We are also grateful to Matt Dwyer and to Tewfik Bultan for using JavaMOP in their classes at the Universities of Nebraska and of California, respectively. We express our thanks also to Eric Bodden for his lead of the static analysis optimization efforts in [40], and to the Tracematches [9, 24], PQL [192], Eagle [30] and RuleR [32] teams for inspiring debates and discussions. The research presented in this paper was generously funded by the NSF grants NSF CCF-0916893, NSF CNS-0720512, and NSF CCF-0448501, by NASA grant NASA-NNL08AA23C, by a Samsung SAIT grant and by several Microsoft gifts and UIUC research board awards.

Sadly, the second author, Feng Chen, passed away on August 8, 2009, in the middle of this project, due to an undetected blood clot. Feng was the main developer of JavaMOP and a co-inventor of most of its underlying techniques and algorithms, including those in this paper. His results and legacy will outlive him. May his soul rest in peace.

see if there is anything relevant at the end of [228], after end-document

Chapter 16

Efficient Formalism-Independent Monitoring of Parametric Properties

Material from [57]. See if it adds anything to the already existing material in the previous section.

Abstract:

Parametric properties provide an effective and natural means to describe object-oriented system behaviors, where the parameters are typed by classes and bound to object instances at runtime. Efficient monitoring of parametric properties, in spite of increasingly growing interest due to applications such as testing and security, imposes a highly non-trivial challenge on monitoring approaches due to the potentially huge number of parameter instances. Existing solutions usually compromise their expressiveness for performance or vice versa. In this paper, we propose a generic, in terms of specification formalism, yet efficient, solution to monitoring parametric specifications. Our approach is based on a general algorithm for slicing parametric traces and makes use of *static* knowledge about the desired property to optimize monitoring. The needed knowledge is not specific to the underlying formalism and can be easily computed when generating monitoring code from the property.

Our approach works with any specification formalism, providing better and extensible expressiveness. Complete proofs for our optimized monitoring algorithm not present in [57] are given. Also, a thorough evaluation shows that our technique out performs other state-of-art techniques optimized for particular logics or properties.

16.1 Introduction

Monitoring executions of a system against expected properties plays an important role not only in different stages of software development, e.g., testing and debugging, but also in the deployed system as a mechanism to increase system reliability. Numerous approaches, such as [126, 165, 88, 58, 28, 9, 25, 192, 118, 62], have been proposed to build effective and efficient monitoring solutions for different applications. More recently, monitoring of parametric specifications, i.e., specifications with free variables, has received increasing interest due to its effectiveness at capturing system behaviors, as shown in the following example about interaction between the classes `Map`, `Collection` and `Iterator` in Java.

`Map` and `Collection` implement data structures for mappings and collections, respectively. `Iterator` is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a `Map` object using `Iterator`. But, since a `Map` object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API specification, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. A *violating* behavior with regards to this property, which we call `UnsafeMapIterator`, can be naturally specified using future time linear temporal logic (FTLTL) with parameters: given that m, c, i are objects of `Map`, `Collection` and `Iterator`, respectively, $\forall m, c, i. \diamond (\text{create_coll}\langle m, c \rangle \wedge \diamond (\text{create_iter}\langle c, i \rangle \wedge \diamond (\text{update_map}\langle m \rangle \wedge \diamond \text{use_iter}\langle i \rangle)))$, where `create_coll` is creating a collection from a map, `create_iter` is creating an iterator from a collection, `update_map` is updating the map, and `use_iter` is using the iterator; \diamond means eventually in the future. The formula describes the following sequence of actions: `Collection` c is obtained from a `Map` m , an iterator i is created from c , m is changed, and then i is accessed. When an observed execution satisfies this

```

UnsafeMapIterator(Map m, Collection c, Iterator i){
  event create_coll after(Map m) returning(Collection c) : (call(* Map.values()) || call(* Map.keySet())) && target(m) {}
  event create_iter after(Collection c) returning(Iterator i) : call(* Collection.iterator()) && target(c) {}
  event use_iter before(Iterator i) : call(* Iterator.next()) && target(i) {}
  event update_map after(Map m) : (call(* Map.remove(..)) || call(* Map.put(..))
    || call(* Map.putAll(..)) || call(* Map.clear())) && target(m) {}

  fsm: start [ create_coll -> s1 ]
        s1 [ update_map -> s1, create_iter -> s2 ]
        s2 [ use_iter -> s2, update_map -> s3 ]
        s3 [ update_map -> s3, use_iter -> end ]
        end []
    @end{ System.out.println("fsm: Accessed Invalid Iterator!"); __RESET; }

  ere : create_coll update_map* create_iter use_iter* update_map update_map* use_iter
    @match{ System.out.println("ere: Accessed Invalid Iterator!"); __RESET; }

  cfg : S -> create_coll Updates create_iter Nexts update_map Updates use_iter,
    Nexts -> Nexts use_iter | epsilon,
    Updates -> Updates update_map | epsilon
    @match{ System.out.println("cfg: Accessed Invalid Iterator!"); __RESET; }

  fttl: <>(create_coll /\ <>(create_iter /\ <>(update_map /\ <>use_iter)))
    @validation{ System.out.println("fttl: Accessed Invalid Iterator!"); __RESET; }

  ptl: use_iter -> ((<*> (create_iter /\ (<*> create_coll))) -> (!update_map) Since create_iter))
    @violation{ System.out.println("ptl: Accessed Invalid Iterator!"); __RESET; }
}

```

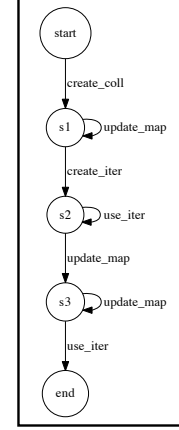


Figure 16.1: FSM, ERE, CFG, FTLTL, and PTLTL UnsafeMapIterator. Inset: graphical depiction of the property.

formula, the UnsafeMapIterator property is broken in the execution.

It is highly non-trivial to monitor such parametric specifications efficiently. We may see a tremendous number of parameter instances during the execution; for example, it is not uncommon to see hundreds of thousands of iterators in one execution. Also, some events may contain partial information about parameters, making it more difficult in locating other relevant parameter bindings during the monitoring process; for example, in the above specification, when a `update_map⟨m⟩` is received, we need to find all `create_coll⟨m, c⟩` events with the same binding for m , and transitively, all `create_iter⟨c, i⟩` with the same c as that `create_coll`.

Several approaches were introduced to support the monitoring of parametric specifications, including Eagle [28], Tracematches [9, 25], PQL [192], PTQL [118]. However, they are all limited in terms of supported specification formalisms or viable execution traces. These techniques follow a formalism-dependent approach, that is, they have their parametric specification formalisms hardwired, e.g., regular patterns (like Tracematches), context-free patterns (like PQL) with parameters, etc., and then develop algorithms to generate monitoring code for the particular formalisms. Although this approach provides a feasible solution to monitoring parametric specifications, we argue that it not only has limited expressiveness, but also

causes unnecessary complexity in developing optimal monitor generation algorithms, often leading to inefficient monitoring. In fact, experiments in [62] and Section 16.7 show that our formalism-independent solution generates more efficient monitoring code than other existing tools.

The system upon which we build in this paper, MOP [62], does not fix the formalism to use in the specification. Instead, MOP provides a generic framework for monitoring of parametric specifications, which allows one to use existing non-parametric formalisms in parametric specifications. Unfortunately, however, the MOP implementation from 2007 [62] for parametric monitoring supports only those specifications in which the first event for any matching trace instantiates all the parameters of the property (the earliest MOP algorithm, published in [58], supported only one parameter). This limitation prevents it from monitoring a large subset of parametric properties, including the above `UnsafeMapIterator` property: the traces specified by `UnsafeMapIterator` begin with `create_coll`, which does not instantiate parameter *i*.

In this paper, we present a general technique to build optimized parametric monitors from non-parametric monitors, following the spirit of MOP but *without limitation*. The presented technique is based on the theoretical results in [65], which was focused on a general, theoretical solution for handling parametric trace and proposed a conceptual algorithm¹. In this novel technique, we apply knowledge about the monitored *property* to improve efficiency. The needed knowledge, encoded as *enable sets*, depends only on the property and not on the formalism in which it is specified. It can be easily computed as a *side effect* when generating a monitor from the property, as discussed in Section 16.5.

Our technique has been implemented in the latest version of JavaMOP². An extensive evaluation shows that the proposed technique not only allows for greater expressiveness, but also significantly improves the efficiency of monitoring in comparison to prior techniques with fixed logical formalisms.

The technique presented here is based on the novel concept of *enable sets*, which abstract necessary information from a property specified in any of the MOP formalisms. This abstraction is sufficient to know which events must occur for a given event to be acceptable in the current state of the

¹Note that the evaluation results in [65], are based on the technique presented in this paper. The technique was only very briefly mentioned in [65], due to its different focus, and because the optimization had not yet been formalized.

²JavaMOP is the Java specialization of MOP, which is itself a framework generic in requirements specification formalisms [62].

monitor. For instance, it is able to abstract the notion that a given object must be created before it can be modified. Using this information, we are able to vastly reduce the number of instance monitors necessary to perform monitoring.

This new technique of optimization based on enable sets, combined with the new general parametric algorithm from [65], represents the first *efficient, modular* technique for monitoring fully general properties (i.e., the properties do not need to instantiate all the parameters in the creation events or use a fixed logical formalism). In fact, it is *more* efficient than the systems that do use a fixed formalism. For the (enable-set-)optimized JavaMOP, only 7 out of 66 of our tested cases caused more than 10% runtime overhead. The numbers for the non-optimized JavaMOP and Tracematches are 9 out of 66 and 15 out of 44, respectively. On two cases the optimized JavaMOP has over an order of magnitude less overhead than Tracematches, and the non-optimized JavaMOP fails to complete the runs, running out of memory. On five other cases, Tracematches has at least twice the overhead of optimized JavaMOP. On any case with noticeable overhead, the enable set optimization produces a notable reduction in overhead.

This paper is an expanded version of [57]. It provides clarifications of the techniques used to optimized generic parametric monitoring above and beyond those of [57], as well as proofs of correctness for algorithm $\mathbb{D}\langle X \rangle$ (see Section 16.6.2).

Contributions. The major contributions of this paper are:

1. A formalism-independent technique for monitoring parametric properties, which overcomes the limitations of existing techniques without reducing performance.
2. A novel concept of enable sets, which encodes static knowledge of the property to monitor and facilitates the optimization of the monitoring process, together with algorithms to compute enable sets for several requirements specification formalisms.
3. Proofs of correctness for the monitoring algorithm $\mathbb{D}\langle X \rangle$ are presented, which were not included in [57].
4. An extensive evaluation and comparison of the proposed solution with Tracematches, an efficient monitoring system for regular expression based properties [9, 25].

This paper expands over [57] by adding detailed proofs of algorithm $\mathbb{D}\langle X \rangle$ and with expanded commentary throughout, including many examples for the more difficult technical concepts.

Outline. The remainder of this paper is as follows: Section 16.2 provides an intuitive overview of our technique, providing an understanding of the more formal parts of the paper that follow. Section 16.3 provides definitions and examples regarding parametric traces. Section 16.4 discusses a modification of the online monitoring algorithm presented in [65] to only create monitor instances for events deemed *monitor creation events*. Section 16.5 presents the *enable sets*, which drive our optimization. Section 16.6 presents our optimized online trace algorithm, with a proof of correctness. Section 16.7 discusses the implementation of the algorithm, and presents the results of our evaluation, which show the efficiency of our technique. Lastly, Section 17.6 concludes the paper.

16.2 Approach Overview

To illustrate our technique we expand the `UnsafeMapIterator` example discussed in Section 17.1. Figure 16.1 shows a JavaMOP specification of the `UnsafeMapIterator` property using five different formalisms: finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), future-time linear temporal logic (FTLTL), and past-time linear temporal logic (PTLTL). Because each of the properties in Figure 16.1 is the same, five messages will be reported whenever an `Iterator` is incorrectly used after an update to the underlying `Map`. We show all five of them to emphasize the formalism-independence of our approach. On the first line, we name the specified property and give the parameters used in the specification. Events are defined using AspectJ [164] pointcut and advice syntax. For example, `create_coll` is defined as the return value of a call to either the function `values` or `keyset` of the `Map` class. We adopt AspectJ syntax to define events in JavaMOP because it is an expressive language for defining observation points in a Java program. As mentioned, every event may instantiate some parameters at runtime. This can be seen in Figure 16.1: `create_coll` will instantiate parameters `m` and `c` using the target and the return value of the method call. When one defines a pattern or formula there are implicit events, which must begin traces; we call these *monitor creation events*. For example, in a pattern language like ERE, the monitor creation events are the first events that appear in the pattern. We assume a

#	Event	#	Event
1	<code>create_coll</code> $\langle m_1, c_1 \rangle$	7	<code>update_map</code> $\langle m_1 \rangle$
2	<code>create_coll</code> $\langle m_1, c_2 \rangle$	8	<code>use_iter</code> $\langle i_2 \rangle$
3	<code>create_iter</code> $\langle c_1, i_1 \rangle$	9	<code>create_coll</code> $\langle m_2, c_3 \rangle$
4	<code>create_iter</code> $\langle c_1, i_2 \rangle$	10	<code>create_iter</code> $\langle c_3, i_4 \rangle$
5	<code>use_iter</code> $\langle i_1 \rangle$	11	<code>use_iter</code> $\langle i_4 \rangle$
6	<code>create_iter</code> $\langle c_2, i_3 \rangle$		

Figure 16.2: Possible execution trace over the events specified in `UnsafeMapIterator`.

semantics where events that occur before monitor creation events are ignored. The crucial point is that this example could not be monitored using the original MOP parametric monitoring algorithm [62] because `create_coll`, the only monitor creation event, does not instantiate the `Iterator` parameter i .

JavaMOP automatically synthesizes AspectJ instrumentation code from the specification, which is weaved into the program we wish to monitor by any standard AspectJ compiler. In this way, executions of the monitored program will produce traces made up of events defined in the specification, as those in Figure 16.1. Consider the example eleven event trace in Figure 16.2 over the events defined in Figure 16.1. The `#` column gives the numbering of the events for easy reference. Every event in the trace starts with the name of the event, e.g., `create_coll`, followed by the parameter binding information, e.g., $\langle m_1, c_1 \rangle$ that binds parameters m and c with a map object m_1 and a collection c_1 , respectively. Such a trace is called a *parametric trace* since it contains events with parameters.

Our approach to monitoring parametric traces against parametric properties is based on the observation that each parametric trace actually contains multiple *non-parametric trace slices*, each for a particular parameter binding instance. The formal definition of the trace slice can be found in Section 16.3, but intuitively, a slice of a parametric trace for a particular parameter binding consists of names of all the events that have identical or *less informative* parameter bindings. Informally, a parameter binding b_1 is identical or less informative than a parameter binding b_2 if and only if the parameters for which they have bindings agree, and b_2 binds either an equal number of parameters or more parameters: parameter $\langle m_1, c_2 \rangle$ is less informative than

Instance	Slice	Status
$\langle m_1 \rangle$	update_map	?
$\langle m_1, c_1 \rangle$	create_coll update_map	?
$\langle m_1, c_2 \rangle$	create_coll update_map	?
$\langle m_2, c_3 \rangle$	create_coll	?
$\langle m_1, c_1, i_1 \rangle$	create_coll create_iter use_iter update_map	?
$\langle m_1, c_1, i_2 \rangle$	create_coll create_iter update_map use_iter	match
$\langle m_1, c_2, i_3 \rangle$	create_coll create_iter update_map	?
$\langle m_2, c_3, i_4 \rangle$	create_coll create_iter use_iter	?

Figure 16.3: Slices for the trace in Figure 16.2.

$\langle m_1, c_2, i_3 \rangle$ because the parameters they both bind, m and c , agree on their values, m_1 and c_2 , respectively, and $\langle m_1, c_2, i_3 \rangle$ binds one more parameter. From here on we will simply say less informative to mean identical or less informative. Figure 16.3 shows the trace slices and their corresponding parameter bindings contained in the trace in Figure 16.2. The Status column denotes the monitor output category that the slice falls into (for ERE). In this case everything but the slice for $\langle m_1, c_1, i_2 \rangle$, which matches the property, is in the “?” (undecided) category. For example, the trace for the binding $\langle m_1, c_1 \rangle$ contains `create_coll update_map` (the first and seventh events in the trace) and the trace for the binding $\langle m_1, c_1, i_2 \rangle$ is `create_coll create_iter update_map use_iter` (the first, fourth, seventh, and eighth events in the trace).

Based on this observation, our approach creates a set of monitor instances during the monitoring process, each handling a trace slice for a parameter binding. Figure 16.4 shows the set of monitor instances created for the trace in Figure 16.2, each monitor labeled by the corresponding parameter binding. This way, the monitor *does not need to handle the parameter information* and can employ any existing technique for ordinary, non-parametric traces, including state machines and push-down automata, providing a formalism-independent way to check parametric properties. When an event comes, our algorithm will dispatch it to related monitors, which will update their states accordingly. For example, the seventh event in Figure 16.2, `update_map` $\langle m_1 \rangle$, will be dispatched to monitors for $\langle m_1, c_1 \rangle$, $\langle m_1, c_2 \rangle$, $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$. New monitor instances will be created if the event contains

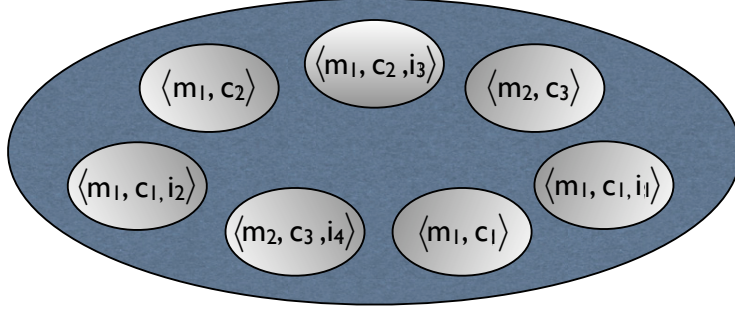


Figure 16.4: A parametric monitor with corresponding parameter instance monitors.

new parameter instances. For example, when the third event in Figure 16.2, `create_iter` $\langle c_1, i_1 \rangle$, is received, a new monitor will be created for $\langle m_1, c_1, i_1 \rangle$ by combining $\langle m_1, c_1 \rangle$ in the first event with $\langle c_1, i_1 \rangle$. Detailed discussion about the monitoring algorithm can be found in Section 16.4.

An algorithm to build parameter instances from observed events, like the one introduced in [65], may create many useless monitor instances leading to prohibitive runtime overheads. For example, Figure 16.3 does not need to contain the binding $\langle m_1, c_3, i_4 \rangle$ even though it can be created by combining the parameter instances of `update_map` $\langle m_1 \rangle$ (the seventh event) and `create_iter` $\langle c_3, i_4 \rangle$ (the tenth event). It is safe to ignore this binding here because m_1 is not the underlying map for c_3, i_4 . It is critical to minimize the number of monitor instances created during monitoring. The advantage is twofold: (1) that it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved since fewer monitors are triggered for each received event.

We present an effective solution in this paper to minimize the created monitors, based on the concept of the *enable set*, which is formally discussed in Section 16.5. An enable set is constructed for each event, say e , defined for a particular property. The enable set associated with e is a set of sets of parameters. Each of these sets of parameters denotes parameters that must have been seen before the arrival of event e , for e to be acceptable by a monitor instance. Consider the event `update_map`, it may occur anywhere in a matching trace, *except* for as the first event. Because the first event must be `create_coll` in a matching trace, and because `create_coll` instantiates both m and c , one of the sets in the enable set for `update_map` must be $\{m, c\}$. However, `update_map` may (in fact, must, to match the pattern) occur after

the `create_iter` event. Because `create_iter` may not occur before `create_coll` we also have the set $\{m, c, i\}$ in the enable set for `update_map`. The final result for the enable set for `update_map` is thus: $\{\{m, c\}, \{m, c, i\}\}$. Therefore, when `update_map` $\langle m_1 \rangle$ arrives (the seventh event), the instance monitors for $\langle m_1, c_1 \rangle$ and $\langle m_1, c_2 \rangle$ must be updated because they bind $\{m, c\}$, and the instance monitors for $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$ must be updated because they bind $\{m, c, i\}$, and have the same value for m (m_1). In this example all of the instances to update have already been created by the time the event arrives, but it should also be noted that no new instances can be created because at least m and c must be bound before `update_map` can occur.

It is worth mentioning that one may reduce the needed monitors using static program analysis, e.g., the one introduced in [40]. However, such techniques are based on the program targeted for monitoring and lead to drawbacks in practice: (1) it is a more complex and thus slower analysis and (2) the analysis must be run for every target program, making the approach non-modular. For example, if the property to monitor is related to some library, one will have to run the analysis for every program using the library, which can be expensive, and often infeasible. The analysis needed by our approach, on the other hand, is usually much quicker³, because properties tend to be much smaller than the programs they are designed to monitor. Moreover, our optimization technique requires no additional analysis when used in a situation, like for a library, where a property is checked for different programs, because the enable set is derived from the property itself instead of the targeted program.

16.3 Background: Parametric Monitoring

In this section, we briefly introduce the semantics of parametric monitoring based on parametric trace slicing. More details, including further formal definitions and proofs, can be found in [65]. We include only the core definitions here to make this paper self-contained.

³The analysis is upper bounded by the number of acyclic paths from the start state/symbol through a finite state machine/context free grammar, because convergence is achieved through one cycle. Finite state machines and context free grammars for properties tend to be small.

16.3.1 Events, Traces and Properties

Traces are sequences of events. Parametric events can carry data-values, as instances of parameters. Parametric traces are traces over parametric events. Properties are trace classifiers, that is, mappings partitioning the space of traces into categories (violating traces, validating traces, ? traces, etc.). Parametric properties are parametric trace classifiers and provide, for each parameter instance, the category to which the trace slice corresponding to that parameter instance belongs. Trace slicing is defined as a *reduct* operation that forgets all the events that are unrelated to the given parameter instance.

Definition 86 *Let \mathcal{E} be a set of (non-parametric) events, called **base events** or simply **events**. An \mathcal{E} -**trace**, or simply a (non-parametric) **trace** when \mathcal{E} is understood, is any finite sequence of events in \mathcal{E} , that is, an element in \mathcal{E}^* . If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

For example, `{create_coll, create_iter, use_iter, update_map}` is the set of (base) events from Figure 16.1, and `create_coll create_iter use_iter update_map` is a (non-parametric) trace.

Definition 87 *An \mathcal{E} -**property** P , or simply a (base or non-parametric) **property**, is a map $P : \mathcal{E}^* \rightarrow \mathcal{C}$ partitioning the set of traces into categories \mathcal{C} .*

It is common, though not enforced, that \mathcal{C} includes *validating*, *violating*, and *don't know* (or *?*) categories, possibly with different names to capture the underlying intuition of the logic (e.g., *match*, like for ERE and CFG). For example, for the regular pattern in Figure 16.1, `create_coll create_iter update_map use_iter` is a *matching* trace, `create_coll create_iter` is a *don't know* trace if the trace is not finished, and `create_coll update_map` is a *violating* trace. In general, \mathcal{C} , the co-domain of P , can be any set (finite or infinite).

Definition 88 *Let X be a set of **parameters** and let V be a set of corresponding **parameter values** (e.g., objects in Java). If \mathcal{E} is a set of events (Definition 86), then let $\mathcal{E}\langle X \rangle$ denote the set of corresponding **parametric events** $e\langle \theta \rangle$, where e is a base event in \mathcal{E} and θ is a **parameter instance**, i.e., an element in $[X \rightarrow V]$, the set of partial maps from X to V . \perp is the empty partial map. A **parametric trace** is a trace with events in $\mathcal{E}\langle X \rangle$, i.e., a word in $\mathcal{E}\langle X \rangle^*$.*

For example, if $X = \{m, c, i\}$ is a set of parameters (of types `{Map, Collection, Iterator}`, respectively) and $V = \{m_1, c_1, i_1, i_2\}$, then `create_coll` $\langle m \mapsto$

$m_1, c \mapsto c_1$, $\text{create_iter}\langle c \mapsto c_1, i \mapsto i_1 \rangle$, and $\text{use_iter}\langle i \mapsto i_1 \rangle$, are parametric events and $\text{create_coll}\langle m \mapsto m_1, c \mapsto c_1 \rangle \text{create_iter}\langle c \mapsto c_1, i \mapsto i_1 \rangle \text{use_iter}\langle i \mapsto i_1 \rangle$ is a parametric trace. In practice, a parametric event usually instantiates a specific set of parameters, which are given in its *event definition*:

Definition 89 Let X be a set of parameters. If \mathcal{E} is a set of base events like in Definition 86, we define a **parametric event definition**, or **event definition** for short, as a function $\mathcal{D}_{\mathcal{E}} : \mathcal{E} \rightarrow \mathcal{P}_f(X)$, where \mathcal{P}_f is “finite power set”, that maps an event to a set of parameters that will be instantiated by e at runtime. Parametric event $e\langle\theta\rangle$ is **$\mathcal{D}_{\mathcal{E}}$ -consistent** if $\text{Dom}(\theta) = \mathcal{D}_{\mathcal{E}}(e)$. Parametric trace τ is **$\mathcal{D}_{\mathcal{E}}$ -consistent** if $e\langle\theta\rangle$ is $\mathcal{D}_{\mathcal{E}}$ -consistent for any $e\langle\theta\rangle \in \tau$.

The example in Figure 16.1 contains the parametric event definition ($\text{create_coll} \mapsto \{m, c\}, \text{create_iter} \mapsto \{c, i\}, \text{use_iter} \mapsto \{i\}, \text{update_map} \mapsto \{m\}$). It states that two parameters, namely, m and c , will be instantiated at runtime when a parametric event $\text{create_coll}\langle\theta\rangle$ is received, etc.; $\text{create_coll}\langle m \mapsto m_1, c \mapsto c_1 \rangle$ is therefore one of its instances. All the parametric traces used in the remaining of this paper are assumed to follow certain given event definitions. Also, from here on we simplify the representation of parametric instances by hiding their domains when they are understood from the context. For example, given the above parametric event definition, we use $\text{create_coll}\langle m_1, c_1 \rangle$ instead of $\text{create_coll}\langle m \mapsto m_1, c \mapsto c_1 \rangle$, and $\langle m_1, c_1 \rangle$ instead of $\langle m \mapsto m_1, c \mapsto c_1 \rangle$.

Definition 90 Parameter instance θ is **compatible with** parameter instance θ' if for any parameter $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible parameter instances θ and θ' , written $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

θ' is **less informative** than θ , written $\theta' \sqsubseteq \theta$, if and only if for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$. \sqsubseteq is a partial order. Recall that we say less informative when we mean less informative.

With the notation above, $\langle m_1, c_1 \rangle$ and $\langle c_1, i_1 \rangle$ are compatible and $\langle m_1, c_1 \rangle \sqcup \langle c_1, i_1 \rangle = \langle m_1, c_1, i_1 \rangle$. Logically, \perp is compatible with, and less informative than, all parameter instances, because it does not bind any parameters.

Definition 91 Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and θ in $[X \rightarrow V]$, we let the θ -**trace slice** $\tau \upharpoonright_\theta \in \mathcal{E}^*$ be the non-parametric trace in \mathcal{E}^* defined as follows:

- $\epsilon \upharpoonright_\theta = \epsilon$, where ϵ is the empty trace/word, and
- $(\tau e\langle \theta' \rangle) \upharpoonright_\theta = \begin{cases} (\tau \upharpoonright_\theta) e & \text{when } \theta' \sqsubseteq \theta \\ \tau \upharpoonright_\theta & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$

Therefore, the trace slice $\tau \upharpoonright_\theta$ first filters out all the parametric events that are not relevant for the instance θ , i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ , it forgets the parameters so that the trace can be checked against base, non-parametric properties. Consider the parametric trace `create_coll` $\langle m_1, c_1 \rangle$ `create_iter` $\langle c_1, i_1 \rangle$ `use_iter` $\langle i_1 \rangle$ `update_map` $\langle m_1 \rangle$ `create_coll` $\langle m_1, c_2 \rangle$. The trace slice for $\langle m_1 \rangle$ is `update_map`, for $\langle m_1, c_1 \rangle$ is `create_coll` `update_map`, for $\langle m_1, c_2 \rangle$ is `create_coll`, and for $\langle m_1, c_1, i_1 \rangle$ is `create_coll` `create_iter` `use_iter` `update_map`.

This definition of trace slicing is designed specifically for monitoring. Given a parametric property to monitor, the parameter set X used in the monitoring process is fixed accordingly. In other words, the monitoring process extracts from the observed execution a parametric trace containing only parameter bindings for parameters in X . Therefore, it is crucial to discard parameter instances that are not relevant to θ during the slicing, even including those more informative than θ , because, otherwise, monitors for incompatible parameter instances may interfere with one another, resulting in incorrect monitoring. For example, if a monitor is created for $\langle m_1, c_1 \rangle$, it should not accept events containing information about i , e.g., `create_iter` $\langle c_1, i_1 \rangle$ and `create_iter` $\langle c_1, i_2 \rangle$. Otherwise, incompatible parameter instances $\langle m_1, c_1, i_1 \rangle$ and $\langle m_1, c_1, i_2 \rangle$ would “interfere” with each other in the parameter instance monitor for $\langle m_1, c_1 \rangle$.

Definition 92 Let X be a set of parameters with their corresponding values V , like in Definition 88, and let $P : \mathcal{E}^* \rightarrow \mathcal{C}$ be a non-parametric property like in Definition 87. Then we define the **parametric property** $\Lambda X . P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and categories $[[X \rightarrow V] \rightarrow \mathcal{C}]$)

$$\Lambda X . P : \mathcal{E}\langle X \rangle^* \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$$

defined as $(\Lambda X . P)(\tau)(\theta) = P(\tau \upharpoonright_\theta)$.

$\Lambda X.P$ is defined as if many instances of P are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

It is worth noting that θ is a partial parameter binding and $\Lambda X.P$ is defined over traces that may not instantiate all the parameters in X . This makes it more expressive than the definition of parametric properties adopted by Tracematches [9, 25], which only supports parametric regular expressions such that *all* the parameters are instantiated whenever the pattern is matched by a trace. For example, consider a property where one wishes to match the start and possible use of a remote resource by a client. A regular expression to match this property would be $\text{start}\langle\text{resource}\rangle \text{ use}\langle\text{client}, \text{resource}\rangle^*$. The trace $\text{start}\langle\text{resource}_1\rangle$ should match the pattern, but it would not be matched in Tracematches, because there is no instantiation of the parameter *client*.

16.3.2 Parametric Monitors

We first define non-parametric monitors M as potentially infinite-state variants of Moore machines; then we define parametric monitors $\Lambda X.M$ as monitors maintaining one non-parametric monitor state per parameter instance.

Definition 93 *A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, \mathbb{B}, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, where S is a set of states, \mathcal{E} is a set of input events, \mathcal{C} is a set of output categories, $\mathbb{B} \in S$ is the initial state, σ is the transition function, and γ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \rightarrow S$ as expected: $\sigma(s, \epsilon) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in \mathcal{E}$, and $w \in \mathcal{E}^*$.*

The above notion of a monitor is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a “by need” basis. Allowing monitors with infinitely many states is a necessity. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on the number of states. For example, monitors for context-free grammars like the ones in [195] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well. What is common to all monitors, though, is that they can take a trace event-by-event and, as each event is processed, classify the observed trace into a category. The following is natural:

Definition 94 $M = (S, \mathcal{E}, \mathcal{C}, \beta, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \rightarrow \mathcal{C}$ iff $\forall w \in \mathcal{E}^*, \gamma(\sigma(\beta, w)) = P(w)$.

We next define parametric monitors: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 95 Given parameters X with corresponding values V and $M = (S, \mathcal{E}, \mathcal{C}, \beta, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, we define the **parametric monitor** $\Lambda X . M$ as the monitor

$$([X \rightarrow V] \rightarrow S], \mathcal{E}\langle X \rangle, [[X \rightarrow V] \rightarrow \mathcal{C}], \lambda \theta. \beta, \Lambda X . \sigma, \Lambda X . \gamma),$$

with $\Lambda X . \sigma : [[X \rightarrow V] \rightarrow S] \times \mathcal{E}\langle X \rangle \rightarrow [[X \rightarrow V] \rightarrow S]$
and $\Lambda X . \gamma : [[X \rightarrow V] \rightarrow S] \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$

defined as for all $\delta \in [[X \rightarrow V] \rightarrow S]$ and all $\theta, \theta' \in [X \rightarrow V]$.

$$\begin{cases} (\Lambda X . \sigma)(\delta, e\langle \theta' \rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\ (\Lambda X . \gamma)(\delta)(\theta) = \gamma(\delta(\theta)) \end{cases}$$

For the sake of rigor, Definition 95 may seem very complicated. All it says, however, is that a state δ of parametric monitor $\Lambda X . M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ , takes parametric events as input, and outputs categories indexed by parameter instances (one output category of M per parameter instance). This is analogous to the intuitive example in Figure 16.3, where the non-parametric slice uniquely determines the state for the parameter instance, and the Status column is the output category. Note that the ? in Status column stands for “don’t know”.

Proposition 41 If M is a monitor for property P then parametric monitor $\Lambda X . M$ is a monitor for parametric property $\Lambda X . P$. (See [65])

This means that we can construct a parametric monitor for a parametric property $\Lambda X . P$ by first creating a non-parametric base monitor for non-parametric property P , and then straight-forwardly extending it as per Definition 95.

In the next sections we discuss algorithms for efficient online monitoring of parametric properties $\Lambda X.P$, given a non-parametric monitor M for property P . We start with a base algorithm that extends algorithm $\mathbb{C}\langle X \rangle$ in [65] to support monitor *creation events*. Then we show that it can be significantly improved provided that enable sets for the property in question are available.

16.4 Monitoring with Creation Events : $\mathbb{C}^+\langle X \rangle$

As mentioned earlier, the monitor creation events are those events that are the first event in matching traces. The point of monitor creation events is to delay the expensive creation of monitor instances until a point where a pattern can actually be matched. For instance, using our example from Figure 16.1, there is no way a trace beginning with `update_map` $\langle m_1 \rangle$ can ever match the patterns or validate the formulae, so creating a monitor instance for m_1 is a waste of both time and memory.

The first challenge to online monitoring of a parametric property is that the state space of potential parameter instances is infinite. Like in [65], we encode partial functions $[[X \rightarrow V] \rightarrow Y]$, which map some parameter instances $[X \rightarrow V]$ to elements in Y , as tables with entries indexed by parameter instances in $[X \rightarrow V]$ and with elements in Y . It can be easily seen that, in what follows, such tables will have a finite number of entries provided that each event instantiates a finite number of parameters, which is always the case.

Figure 16.5 shows the algorithm $\mathbb{C}^+\langle X \rangle$ for online monitoring of parametric property $\Lambda X.P$, given that M is a monitor for P . The algorithm shows which actions to perform, e.g., creating a new monitor state and/or updating the state of related monitors, when an event is received. It is a slightly different variant of algorithm $\mathbb{C}\langle X \rangle$ in [65]. $\mathbb{C}^+\langle X \rangle$ is justified and motivated by experience with implementing and evaluating $\mathbb{C}\langle X \rangle$ in [65], mainly by the following observation: one often chooses to start monitoring at the witness of a specific set of events (instead of monitoring from the beginning of the program). For example, when we monitor the property in Figure 16.1, we can choose to start monitoring on a pair of `m` and `c` objects, (m_1, c_1) , only when a `create_coll` event is received, ignoring all the `update_map` $\langle m_1 \rangle$ events before the creation. We call such events that lead to creation of new monitor states (*monitor creation events*). Algorithm $\mathbb{C}^+\langle X \rangle$ extends $\mathbb{C}\langle X \rangle$ in [65] to support creation events. It is easy to see that $\mathbb{C}\langle X \rangle$ can be regarded as a

```

Algorithm  $\mathbb{C}^+\langle X \rangle(M = (S, \mathcal{E}, \mathcal{C}, \beta, \sigma, \gamma))$ 
Globals: mapping  $\Delta : [[X \rightarrow V] \rightarrow S]$ 
         mapping  $\mathcal{U} : [X \rightarrow V] \rightarrow \mathcal{P}_f([X \rightarrow V])$ 
Initialization:  $\mathcal{U}(\theta) \leftarrow \emptyset$  for any  $\theta \in [X \rightarrow V]$ 

function main( $e\langle\theta\rangle$ )
1  if  $\Delta(\theta)$  undefined then
2      foreach  $\theta_m \sqsubset \theta$  (in reversed topological order) do
3          if  $\Delta(\theta_m)$  defined then goto 5 endif
4      endfor
5      if  $\Delta(\theta_m)$  defined then defineTo( $\theta, \theta_m$ )
6      elseif  $e$  is a creation event then defineNew( $\theta$ ) endif
7      foreach  $\theta_m \sqsubset \theta$  (in reversed topological order) do
8          foreach  $\theta_{comp} \in \mathcal{U}(\theta_m)$  compatible with  $\theta$  do
9              if  $\Delta(\theta_{comp} \sqcup \theta)$  undefined then defineTo( $\theta_{comp} \sqcup \theta, \theta_{comp}$ ) endif
10             endfor
11         endfor
12     endif
13     foreach  $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$  do  $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$  endfor
function defineNew( $\theta$ )
1   $\Delta(\theta) \leftarrow \beta$ 
2  foreach  $\theta'' \sqsubset \theta$  do  $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$  endfor
function defineTo( $\theta, \theta'$ )
1   $\Delta(\theta) \leftarrow \Delta(\theta')$ 
2  foreach  $\theta'' \sqsubset \theta$  do  $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$  endfor

```

Figure 16.5: Monitoring Algorithm $\mathbb{C}^+\langle X \rangle$.

Event	$\text{update_map}\langle m_1 \rangle$	$\text{create_coll}\langle m_1, c_1 \rangle$
Δ	\emptyset	$\langle m_1, c_1 \rangle : \sigma(\beta, \text{create_coll})$
\mathcal{U}	\emptyset	$\perp : \langle m_1, c_1 \rangle$ $\langle m_1 \rangle : \langle m_1, c_1 \rangle$ $\langle c_1 \rangle : \langle m_1, c_1 \rangle$
Event	$\text{create_coll}\langle m_2, c_2 \rangle$	$\text{create_iter}\langle c_1, i_1 \rangle$
Δ	$\langle m_1, c_1 \rangle : \sigma(\beta, \text{create_coll})$ $\langle m_2, c_2 \rangle : \sigma(\beta, \text{create_coll})$	$\langle m_1, c_1 \rangle : \sigma(\beta, \text{create_coll})$ $\langle m_2, c_2 \rangle : \sigma(\beta, \text{create_coll})$ $\langle m_1, c_1, i_1 \rangle : \sigma(\sigma(\beta, \text{create_coll}), \text{create_iter})$
\mathcal{U}	$\perp : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle$ $\langle m_1 \rangle : \langle m_1, c_1 \rangle$ $\langle c_1 \rangle : \langle m_1, c_1 \rangle$ $\langle m_2 \rangle : \langle m_2, c_2 \rangle$ $\langle c_2 \rangle : \langle m_2, c_2 \rangle$	$\perp : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle c_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2 \rangle : \langle m_2, c_2 \rangle$ $\langle c_2 \rangle : \langle m_2, c_2 \rangle$ $\langle i_1 \rangle : \langle m_1, c_1, i_1 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle m_1, c_1 \rangle : \langle m_1, c_1, i_1 \rangle$ $\langle m_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$ $\langle c_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$

Figure 16.6: Sample run of $\mathbb{C}^+\langle X \rangle$. The first row gives the received events; the second and the third rows give the content of Δ and \mathcal{U} , respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., $\sigma(\beta, \text{create_coll})$ represents the state after the event create_coll .

special case of $\mathbb{C}^+\langle X \rangle$, when all the events are creation events. Note that [65] used creation events in the evaluation, but they were not formalized in the algorithm. The proof of $\mathbb{C}^+\langle X \rangle$ is tedious, but is easily derived from the proof of $\mathbb{C}\langle X \rangle$ in [65].

Two mappings are used: Δ and \mathcal{U} . Δ stores the monitor states for parameter instances, and \mathcal{U} maps a parameter instance θ to *all the parameter instances* that have been defined and are properly more informative than θ . In what follows, “the monitor state for θ ” refers to $\Delta(\theta)$ to facilitate reading in some contexts, and, accordingly, “to create a parameter instance θ ” and “to create a monitor state for parameter instance θ ” have the same meaning: to define $\Delta(\theta)$.

When parametric event $e\langle\theta\rangle$ arrives, the algorithm first checks whether θ has been encountered yet by checking if its corresponding monitor state, i.e., $\Delta(\theta)$, has been defined (line 1 in **main**). If θ is encountered for the first time, new parameter instances may need be created. In such a case, we first try to locate the maximum parameter instance (θ_m) which is less informative than θ and for which a monitor state has been created (lines 2 - 4). If such θ_m is found, its monitor state is used to initialize the monitor state for θ (lines 5); otherwise, a new monitor state is created for θ *only if* e is a creation event (line 6). Also, new parameter instances can be created by combining θ with existing parameter instances that are compatible with θ , i.e., they do not have conflicting parameter bindings. An observation here is that if parameter instance θ_{comp} has been created and is compatible with θ then θ_{comp} can be found in $\mathcal{U}(\theta_m)$ for some $\theta_m \sqsubset \theta$ according to the definition of \mathcal{U} . Therefore, algorithm $\mathbb{C}^+\langle X \rangle$ searches through all the $\theta_m \sqsubset \theta$ to find all possible θ_{comp} , examining whether any new parameter instance should be created (lines 7 - 11).

If θ has been seen before, or otherwise after all the new monitor states have been created/initialized as explained above, algorithm $\mathbb{C}^+\langle X \rangle$ invokes all the monitors that need to process e , namely, those whose corresponding parameter instances are more informative than or equal to θ , note that the updates make use of the sets stored in \mathcal{U} to know which instances are more informative (line 13). There are two auxiliary functions: **defineNew** and **defineTo**. The former initializes a new monitor state for the input parameter instance and the latter creates a monitor state for the first input parameter instance using the monitor state for the second instance. Both functions add θ to the sets in table \mathcal{U} for the bindings less informative than θ .

We next use an example run, illustrated in Figure 16.6, to show how

$\mathbb{C}^+\langle X \rangle$ works. In Figure 16.6, we show the contents of Δ and \mathcal{U} after every event (given in the first row of the table) is processed. The observed trace is `update_map` $\langle m_1 \rangle$ `create_coll` $\langle m_1, c_1 \rangle$ `create_coll` $\langle m_2, c_2 \rangle$ `create_iter` $\langle c_1, i_1 \rangle$. We assume that `create_coll` is the only creation event.

The first event, `update_map` $\langle m_1 \rangle$, is not a creation event and nothing is added to Δ and \mathcal{U} . The second event, `create_coll` $\langle m_1, c_1 \rangle$, is a creation event. So a new monitor state is defined in Δ for $\langle m_1, c_1 \rangle$, which is also added to the lists in \mathcal{U} for \perp , $\langle m_1 \rangle$ and $\langle c_1 \rangle$. Note that \perp is less informative than any other parameter instances. The third event `create_coll` $\langle m_2, c_2 \rangle$ is another creation event, incompatible with the second event. Hence, only one new monitor state is added to Δ . \mathcal{U} is updated similarly. The last event `create_iter` $\langle c_1, i_1 \rangle$ is not a creation event. Therefore, no monitor instance is created for $\langle c_1, i_1 \rangle$. It is compatible with the existing parameter instance $\langle m_1, c_1 \rangle$ introduced by the second event but not compatible with $\langle m_2, c_2 \rangle$ due to the conflict binding on the c parameter. The compatible instance $\langle m_1, c_1 \rangle$ can be found from the list for $\langle c_1 \rangle$ in \mathcal{U} . Therefore, a new monitor instance is created for the combined parameter instance $\langle m_1, c_1, i_1 \rangle$ using the state for $\langle m_1, c_1 \rangle$ in Δ . \mathcal{U} is also updated to add the combined parameter instance into the lists of parameter instances that are less informative.

16.5 Limitations of $\mathbb{C}^+\langle X \rangle$ and Enable Sets

$\mathbb{C}^+\langle X \rangle$ does not make any assumption on the given monitor M . In other words, one may monitor properties written in any specification formalism, e.g., ERE, CFG, PTLTL etc., as long as one also provides a monitor generation algorithm for said formalism. However, this generality leads to extra monitoring overhead in some cases. Thus we introduce our novel optimization based on the concept of *enable sets*.

To motivate the optimization, let us continue the run in Figure 16.6 to process one more event, `use_iter` $\langle i_1 \rangle$. The result is shown in Figure 16.7. `use_iter` $\langle i_1 \rangle$ is not a creation event and no monitor instance is created for $\langle i_1 \rangle$. Since $\langle i_1 \rangle$ is compatible with $\langle m_2, c_2 \rangle$, a new monitor instance is defined for $\langle m_2, c_2, i_1 \rangle$. The monitor instance for $\langle m_1, c_1, i_1 \rangle$ is then updated according to `use_iter` because $\langle i_1 \rangle$ is less informative than $\langle m_1, c_1, i_1 \rangle$. \mathcal{U} is also updated to add $\langle m_2, c_2, i_1 \rangle$ to the lists for all the parameter instances less informative than $\langle m_2, c_2, i_1 \rangle$. New entries are added into \mathcal{U} during the update since some of the less informative parameter instances, e.g., $\langle m_2, i_1 \rangle$, have not been used before this event.

Event	$\text{use_iter}\langle i_1 \rangle$
Δ	$\langle m_1, c_1 \rangle : \sigma(\beta, \text{create_coll})$ $\langle m_2, c_2 \rangle : \sigma(\beta, \text{create_coll})$ $\langle m_1, c_1, i_1 \rangle : \sigma(\sigma(\sigma(\beta, \text{create_coll}), \text{create_iter}), \text{use_iter})$ $\langle m_2, c_2, i_1 \rangle : \sigma(\sigma(\beta, \text{create_coll}), \text{use_iter})$
\mathcal{U}	$\perp : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle c_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2 \rangle : \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle c_2 \rangle : \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle i_1 \rangle : \langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2, c_2 \rangle : \langle m_2, c_2, i_1 \rangle$ $\langle m_2, i_1 \rangle : \langle m_2, c_2, i_1 \rangle$ $\langle c_2, i_1 \rangle : \langle m_2, c_2, i_1 \rangle$ $\langle m_1, c_1 \rangle : \langle m_1, c_1, i_1 \rangle$ $\langle m_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$ $\langle c_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$

Figure 16.7: Following the run of Fig. 16.6.

Creating the monitor instance for $\langle m_2, c_2, i_1 \rangle$ is needed for the correctness of $\mathbb{C}^+\langle X \rangle$, but it can be avoided when more information about the program or the specification is available. For example, according to the semantics of `Iterator`, no event `create_iter` $\langle c_2, i_1 \rangle$ will occur in the following execution since an iterator can be associated to only one collection. Hence, the monitor for $\langle m_2, c_2, i_1 \rangle$ will never reach the validation state and we do not need to create it from the beginning. However, such semantic information about the program is very difficult to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary monitor creations by analyzing the specification to monitor.

When monitoring a program against a specific property, usually only a certain subset of property categories, (\mathcal{C} in Definition 87), is checked. For example, in Figure 16.1, the regular expression specifies a defective interaction among related `Map`, `Collection` and `Iterator` objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for a parameter instance of parameter set $\{m, c, i\}$, `create_coll` and `create_iter` should be observed before `use_iter` is encountered for the first time in monitoring. Otherwise, the trace slice for $\{m, c, i\}$ will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_2, c_2, i_1 \rangle$ in Figure 16.7 is not needed. When event `use_iter` $\langle i_1 \rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, c_2 \rangle$ exists without the monitor state for $\langle m_2, c_2, i_1 \rangle$, like in Figure 16.7, it can be inferred that in the trace slice for $\langle m_2, c_2, i_1 \rangle$ only events `create_coll` and/or `update_map` could have occur before `use_iter`. If `create_iter` has also occurred before `use_iter`, the monitor state for $\langle m_2, c_2, i_1 \rangle$ would have been created previously. Therefore, we can infer, when event `use_iter` $\langle i_1 \rangle$ is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_2, c_2, i_1 \rangle$, that is to say, the monitor for $\langle m_2, c_2, i_1 \rangle$ will never reach the validation state.

This observation shows that the knowledge about the specified property can be applied to avoid unnecessary creation of monitor states. This way, the sizes of Δ and \mathcal{U} can be reduced, reducing the monitoring overhead. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism, and that it can be computed easily. How this information is used is discussed in Section 16.6.

16.5.1 Enable Sets

Definition 96 Given $\tau \in \mathcal{E}^*$ and $e, e' \in \tau$, we let $e' \leadsto_\tau e$ denote that e' occurs before the first occurrence of e in τ . Let $\text{enable}_\tau : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{E})$ be the **trace enable set** of $e \in \mathcal{E}$, which is the function defined as: $\text{enable}_\tau(e) = \{e' \mid e' \leadsto_\tau e\}$.

Note that if $e \notin \tau$ then $\text{enable}_\tau(e) = \emptyset$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event e is encountered during monitoring but some event $e' \in \text{enable}_\tau(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace τ when it finishes. This observation can be extended to check, before an execution finishes, whether the execution can generate a trace belonging to some designated property categories. The designated property categories are called the *goal* of the monitoring in what follows.

Definition 97 Given $P : \mathcal{E}^* \rightarrow \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the **property enable set** is defined as a function $\text{enable}_\mathcal{G}^\mathcal{E} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\text{enable}_\mathcal{G}^\mathcal{E}(e) = \{\text{enable}_\tau(e) \mid P(\tau) \in \mathcal{G}\}$.

Intuitively, if event e is encountered during monitoring but none of event sets $\text{enable}_\mathcal{G}^\mathcal{E}(e)$ has been completely observed, the (incomplete) execution being monitoring will not produce a trace τ s.t. $P(\tau) \in \mathcal{G}$. For example, given the property specified by the ERE-based property in Figure 16.1, where \mathcal{G} contains only the match category, Figure 16.8 shows the property enable set for `UnsafeMapIterator`.

The property enable set provides a sound and fast way to say whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with `create_coll use_iter`, it will never reach the `match` category, because $\{\text{create_coll}\} \notin \text{enable}_\mathcal{G}^\mathcal{E}(\text{use_iter})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is `create_coll` $\langle m_1, c_1 \rangle$ `use_iter` $\langle i_1 \rangle$. At the second event, `use_iter` $\langle i_1 \rangle$, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1 \rangle$, and a monitor state s will be created for $\langle m_1, c_1, i_1 \rangle$ if algorithm $\mathbb{C}^+\langle X \rangle$ is applied. However, since the trace slice for s is `create_coll use_iter`, we can immediately know that s cannot reach the `match` state, and thus there is no need to create and maintain s during monitoring if `match` is the target category.

Event	$\text{enable}_{\mathcal{G}}^{\mathcal{E}}(\text{Event})$
create_coll	$\{\emptyset\}$
create_iter	$\{\{\text{create_coll}\},$ $\{\text{create_coll}, \text{update_map}\}\}$
use_iter	$\{\{\text{create_coll}, \text{create_iter}\},$ $\{\text{create_coll}, \text{create_iter}, \text{update_map}\}\}$
update_map	$\{\{\text{create_coll}\},$ $\{\text{create_coll}, \text{create_iter}\},$ $\{\text{create_coll}, \text{create_iter}, \text{use_iter}\}\}$

Figure 16.8: Property enable set ($\text{enable}_{\mathcal{G}}^{\mathcal{E}}$) for UnsafeMapIterator.

A direct application of the above idea to optimize $\mathbb{C}^+\langle X \rangle$ requires maintaining observed events for every created monitor and comparing event sets when a new parameter instance is found, reducing the improvement of performance. Therefore, we extend the notion of the enable set to be based on parameter sets instead of event sets.

Definition 98 *Given a property $P : \mathcal{E}^* \rightarrow \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters X and a parameter definition $\mathcal{D}_{\mathcal{E}}$, the **property parameter enable set** of event $e \in \mathcal{E}$ is defined as a function $\text{enable}_{\mathcal{G}}^X : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\text{enable}_{\mathcal{G}}^X(e) = \{\cup\{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \text{enable}_{\tau}(e)\} \mid P(\tau) \in \mathcal{G}\}$.*

From now on, we use “enable set” to refer to “property parameter enable set”. For example, given the ERE-based property in Figure 16.1 and $\mathcal{G} = \{\text{match}\}$; Figure 16.9 shows the parameter enable set for UnsafeMapIterator. Then, given again the trace $\text{create_coll}\langle m_1, c_1 \rangle \text{ use_iter}\langle i_1 \rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1 \rangle$ since the parameter set of the paramter instance used to initialize the new monitor state, namely, $\{m, c\}$, is not in $\text{enable}_{\mathcal{G}}^X(\text{use_iter})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin in question. This allows the plugins to remain totally parameter agnostic.

Event	$\text{enable}_{\mathcal{G}}^X(\text{Event})$
create_coll	$\{\emptyset\}$
create_iter	$\{\{m, c\}\}$
use_iter	$\{\{m, c, i\}\}$
update_map	$\{\{m, c\}, \{m, c, i\}\}$

Figure 16.9: Parameter enable set ($\text{enable}_{\mathcal{G}}^X$) for UnsafeMaplterator.

The following result guarantees the correctness of this approach:

Proposition 42 *When algorithm $\mathbb{C}^+\langle X \rangle$ receives event $e\langle\theta\rangle$, if we use $\Delta(\theta')$ to define $\Delta(\theta \sqcup \theta')$ and $\text{Dom}(\theta') \notin \text{enable}_{\mathcal{G}}^X(e)$, then $\Delta(\theta \sqcup \theta') \notin \mathcal{G}$ during the whole monitoring process.*

Proof: The proof follows by contradiction. Assume that $\text{Dom}(\theta') \notin \text{enable}_{\mathcal{G}}^X(e)$ but $\Delta(\theta \sqcup \theta') \in \mathcal{G}$. From Definition 94 and the proof of algorithm $\mathbb{C}\langle X \rangle$ in [65]⁴ it must be the case that there are some $w, w' \in \mathcal{E}^*$ and some $e \in \mathcal{E}$ such that $\mathcal{D}_{\mathcal{E}}(e) = \text{Dom}(\theta)$, $\Delta(\theta') = \sigma(\beta, w)$, $\Delta(\theta \sqcup \theta') = \sigma(\beta, we)$, and $\gamma(\sigma(\beta, wew')) = P(wew') \subseteq \mathcal{G}$. Then from Definitions 96, 97, and 98 it follows that there must be a set of events $\mathcal{E}_w \subseteq \mathcal{E}$ such that each $e' \in \mathcal{E}_w$ must occur at least once in trace w and $\bigcup \{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \mathcal{E}_w\} = \text{Dom}(\theta') \in \text{enable}_{\mathcal{G}}^X(e)$. This violates our initial assumptions, thus a contradiction is found. \square

16.5.2 Computing Enable Sets

The definition of the enable set is general and does not depend on a specific formalism to write the property. However, computing the enable set from a specified property requires understanding of the used formalism. It can be achieved as a “side-effect” of the monitor generation process, in which full knowledge about the property is available.

Case 1: FSM. The algorithm in Figure 16.10 computes the property enable sets for a finite state machine. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, PTLTL, and FTLTL. The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \rightarrow S, F \subseteq S)$. \mathcal{E} is the alphabet, traditionally listed as Σ but changed for consistency, since the alphabets

⁴Recall that the proof from [65] is trivially extended to $\mathbb{C}^+\langle X \rangle$.

```

Algorithm  $\mathcal{EN}_{fsm}(FSM = (\mathcal{E}, S, s_0, \delta, F))$ 
Globals: mapping  $\mathcal{V}_\mu : S \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ 
         mapping  $\text{enable}_G^\mathcal{E} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ 
         set  $R \subseteq S$ 

Initialization: fix  $G \subseteq S$ , compute  $R = \text{reach}(G)$ 

function main()
1  recurse( $s_0, \emptyset$ )
recurse( $s, \mu$ )
1  foreach  $e \in \mathcal{E}$  do
2      if  $\delta(s, e) \in R$  then
3           $\text{enable}_G^\mathcal{E}(e) \leftarrow \text{enable}_G^\mathcal{E}(e) \cup \{\mu - e\}$ 
4      endif
5      let  $\mu' \leftarrow \mu \cup \{e\}$ 
6      if  $\mu' \notin \mathcal{V}_\mu(s)$ 
7           $\mathcal{V}_\mu(s) \leftarrow \mathcal{V}_\mu(s) \cup \{\mu'\}$ 
8          recurse( $\delta(s, e), \mu'$ )
9      endif
10 endfor

```

Figure 16.10: FSM $\text{enable}_G^\mathcal{E}$ computation algorithm.

of our FSMs are event sets. s_0 is the start state, corresponding to β in the definition of a monitor. δ is the transition function, taking a state and an event and mapping to a next state for the machine. F is the set of accept states. In the initialization we compute goal reachability set S by fixing a goal G as an arbitrary set of states, such as the error state for violation, or accept states for matching a pattern originally specified as an ERE. More specifically, G is the subset of S corresponding to the subset of \mathcal{G} in which we are interested. For state $s \in S$, $s \in R$ if and only if there is a path from s to some $s' \in G$. It is computed using a straight-forward depth first search from the initial state. \mathcal{V}_μ is a mapping from states to sets of events; it is used to check for algorithm termination. $\text{enable}_\mathcal{G}^\mathcal{E}$ is the output property enable set, which is converted into a parameter enable set by JavaMOP.

Function `recurse` is first called with $\mu = \emptyset$ and the initial state s_0 (line 1 of `main`). If we think of the FSM as a graph, μ represents the set of edges we have seen at least once in a traversal. For each event in \mathcal{E} (line 1), we check to see if the next state, computed by $\delta(s, e)$ reaches our goal (line 2). If it does, that means we have seen a viable prefix set. From the definition of $\text{enable}_\mathcal{G}^\mathcal{E}$, we know we need to add this prefix set to $\text{enable}_\mathcal{G}^\mathcal{E}$ for the event e , which we do on line 3. Also on line 3, we make sure that we remove e from μ , as an enable set for e is not supposed to contain e . Line 5 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as e . The map \mathcal{V}_μ tells us which μ have been seen in previous recursive steps, in a given state. If a μ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 6 ensures that we only call the recursive step on line 8, if new information can be added. Line 7 keeps \mathcal{V} consistent. Thus the algorithm terminates only when every viable μ has been seen in every reachable state, effectively computing a fixed point.

Case 2: CFG. We also provide an algorithm to compute the enable set for a context-free pattern, which has an infinite monitor state space, as briefly explained in what follows⁵. This is a modification of the algorithm in Figure 16.10.

Let $\mathcal{G} = \{\text{match}\}$. For $\text{enable}_\mathcal{G}^\mathcal{E}$ and a given context-free grammar $G = (NT, \mathcal{E}, P, S)$ we begin with all productions $S \rightarrow \gamma$ and the set $\mu_0 = \emptyset \in \mathcal{P}_f(\mathcal{E})$. For each production, we investigate each $s \in \gamma$ (where \in is, by abuse of notation, used to denote a symbol in a right hand side) from left to

⁵We assume a certain familiarity with context free patterns; definitions can be found in [195], together with explanations on CFG monitoring.

right. If $s \in \mathcal{E}$ we add μ_i to $\text{enable}_{\mathcal{G}}^{\mathcal{E}}(s)$, thus if s is the first symbol in γ we add μ_0 . We then add s to μ_i forming μ_{i+1} . If $s \in NT$ we recursively invoke the algorithm, but rather than use μ_0 , we use μ_i , and each production investigated will be of the form $s \rightarrow \gamma$. We keep track of which $s \in NT$ have been processed, to ensure termination.

Discussion. The general definition of the enable set allows us to separate the concerns of generating efficient monitoring code. On the framework level, such as the algorithms discussed in this paper, we can focus on applying the information encoded in the enable set to generate an efficient monitoring process for parametric properties, while on the logic level, where a monitor is generated for a given non-parametric property written in a specific formalism, one can focus on creating the fastest monitor that verifies the input trace against the property and also on producing the enable set information. The enable set represents static information about the given property and only need be generated once. As mentioned, the static analysis presented in [40], while effective, requires a complex analysis of the target program, which must be performed for every program one wants to monitor.

Other possibilities for optimization are exhibited in the example in Figure 16.7. We discuss two of them here. The first is to make use of the semantics of the program. In this example, we know that an i object is created from a c object and does not relate to other c objects. Hence, we can avoid creating a combination of $\langle m_2, c_2 \rangle$ and $\langle i_1 \rangle$ because i_1 is created from c_1 . However, such semantic information is very difficult to achieve automatically and may require human input. The enable set, on the contrary, can be easily computed by statically analyzing the specification without analyzing any program or human interferences; indeed, the specified property already indicates some semantics of the involved parameters. Nevertheless, we believe that static analysis on the program to monitor, such as that in [40], can and should be applied in conjunction with enable sets to further reduce the monitoring overhead, whenever it is feasible.

Other optimizations are based on heuristics. One reasonable heuristic which can be applied here is that we may only combine parameter instances that are connected to one another through some events which have been observed (we cannot rely on future events in online monitoring). For example, $\langle i_1 \rangle$ and $\langle m_1, c_1 \rangle$ need to be combined to build a new parameter instance because c_1 and i_1 are connected in the second event, `create_coll` $\langle m_1, c_1 \rangle$, in Figure 16.7, but $\langle i_1 \rangle$ and $\langle m_2, c_2 \rangle$ should not be combined due to the heuristic. The intuition is that if two parameter instances do not interact in

any event, it may imply that they are not relevant to each other even if they are compatible. However, because no information about future events is available, such a heuristic can break, for example, when an event connecting the two parameter instances comes afterward. The enable set provides a sound optimization, and we believe that it performs as well as, if not better than, such heuristics in most cases.

16.6 Monitoring with Enable Sets: $\mathbb{D}\langle X \rangle$

In this section we integrate the concept of enable sets with algorithm $\mathbb{C}^+\langle X \rangle$, to improve performance and memory usage. To ease reading, all proofs related to this algorithm can be found in Section 16.6.2.

Given a set of desired value categories \mathcal{G} , Proposition 42 guarantees that we can omit creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to \mathcal{G} . However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in \mathcal{G} can be reported to belong to \mathcal{G} . Let us consider the following example. We monitor to find matching of a regular pattern e_1e_3 and the event definition is $(e_1 \mapsto \{P_1\}, e_2 \mapsto \{P_2\}, e_3 \mapsto \{P_1, P_2\})$ the observed trace is $e_1\langle p_1 \rangle e_2\langle p_2 \rangle e_3\langle p_1, p_2 \rangle$. Also, suppose e_1 is the only creation event. Obviously, the trace does not match the pattern. Figure 16.11 shows the run using the optimization based on the enable set. Only the content of Δ is given for simplicity. At e_1 , a monitor state is created for $\langle p_1 \rangle$ since it is the creation event. At e_2 , no action is taken since $\text{enable}_{\mathcal{G}}^X(e_2) = \emptyset$. At e_3 , a monitor state will be created for $\langle p_1, p_2 \rangle$ using the monitor state for $\langle P_1 \mapsto p_1 \rangle$ since $\text{enable}_{\mathcal{G}}^X e_3 = \{P_1\}$. This way, e_2 is forgotten and a match of the pattern is reported even though it is not correct to do so.

Event	$e_1\langle p_1 \rangle$	$e_2\langle p_2 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$ $\langle p_1, p_2 \rangle : \sigma(\sigma(i, e_1), e_3)$

Figure 16.11: Unsound usage of $\text{enable}_{\mathcal{G}}^X$.

16.6.1 Timestamping Monitors: Algorithm $\mathbb{D}\langle X \rangle$

To avoid unsoundness, we introduce the notion of **disable** stamps of events. $\text{disable} : [[X \rightarrow V] \rightarrow \text{integer}]$ maps a parameter instance to an integer timestamp. $\text{disable}(\theta)$ gives the time when the last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \rightarrow V] \rightarrow \text{integer}]$. \mathcal{T} maps a parameter instance for which a monitor state is defined to the time when the original monitor state is created from a creation event. Specifically, if a monitor state for θ is created using the initial state when a creation event is received (i.e., using the **defineNew** function in algorithm $\mathbb{C}^+\langle X \rangle$), $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for θ is created from the monitor state for θ' , $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Figure 16.12 shows the evolution of **disable** and \mathcal{T} while processing the trace in Figure 16.11.

disable and \mathcal{T} can be used together to track “skipped events”: when a monitor state for θ is created using the monitor state for θ' , if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubset \theta'$ and $\text{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for θ does not belong to the desired value categories \mathcal{G} . Intuitively, $\text{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for θ' was created. But θ'' was not taken into account ($\theta'' \not\sqsubset \theta'$). The only possibility is that e is omitted due to the enable set and thus the trace slice for θ does not belong to \mathcal{G} according to the definition of the enable set. Therefore, in Figure 16.12, no monitor instance is created for $\langle p_1, p_2 \rangle$ at e_3 because $\text{disable}(\langle p_2 \rangle) > \mathcal{T}(\langle p_1 \rangle)$.

Event	$e_1\langle p_1 \rangle$	$e_2\langle p_2 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_1 \rangle : \sigma(i, e_1)$
\mathcal{T}	$\langle p_1 \rangle : 1$	$\langle p_1 \rangle : 1$	$\langle p_1 \rangle : 1$
disable	$\langle p_1 \rangle : 2$	$\langle p_1 \rangle : 2$ $\langle p_2 \rangle : 3$	$\langle p_1 \rangle : 2$ $\langle p_2 \rangle : 3$ $\langle p_1, p_2 \rangle : 4$

Figure 16.12: Sound monitoring using $\text{enable}_{\mathcal{G}}^X$ and timestamps.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using

timestamps. First, if the skipped event is not a creation event, it does not affect the soundness of the algorithm to omit the event because of the definition of creation events. In the above example, if the observed trace is $e_2\langle p_2\rangle e_1\langle p_1\rangle e_3\langle p_1, p_2\rangle$, we will ignore e_2 and report the matching at e_3 since e_1 is the only creation event. The situation becomes more sophisticated when the skipped event is a creation event. For example, we assume that both e_1 and e_2 are creation events in the above example. Figure 16.13 then shows the monitoring process for the parametric trace $e_2\langle p_2\rangle e_1\langle p_1\rangle e_3\langle p_1, p_2\rangle$.

At e_2 , $\Delta(\langle p_2\rangle)$ is defined because it is a creation event. At e_1 , $\Delta(\langle p_1\rangle)$ is defined, but no monitor state is created for $\langle p_1, p_2\rangle$ because $\{P_2\} \notin \text{enable}_{\mathcal{G}}^X(e_1)$. At e_3 , we cannot use $\Delta(\langle p_2\rangle)$ to define $\Delta(\langle p_1, p_2\rangle)$ since $\text{disable}(\langle p_1\rangle) > \mathcal{T}(\langle p_2\rangle)$. Moreover, we cannot use $\Delta(\langle p_1\rangle)$ to define $\Delta(\langle p_1, p_2\rangle)$, either, because $\Delta(\langle p_2\rangle)$ was defined before $\Delta(\langle p_1\rangle)$ but was not used to create $\Delta(\langle p_1, p_2\rangle)$ at e_1 due to the use of the enable set, indicating that the trace slice for $\langle p_1, p_2\rangle$ does not belong to \mathcal{G} , and it should be ignored during monitoring. This intuition can be captured as the following condition: $\mathcal{T}(\langle p_2\rangle) < \mathcal{T}(\langle p_1\rangle)$ and $\langle p_2\rangle \not\sqsubseteq \langle p_1\rangle$. To reiterate, if $\Delta(\theta')$ is used to define $\Delta(\theta)$ and there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$, then the trace slice for θ does not belong to the desired category set \mathcal{G} , because θ would have been in the enable set of θ' if it were in \mathcal{G} . Such a situation happens at the following conditions: 1) a creation event, $e\langle\theta''\rangle$, is encountered before $\Delta(\theta')$ is defined at event e' ; 2) e is omitted when $\Delta(\theta')$ is defined (otherwise $\Delta(\theta'' \sqcup \theta')$ should have been defined and should be used to define θ instead of θ'). The second condition implies that $\text{Dom}(\theta'') \notin \text{enable}_{\mathcal{G}}^X(e')$. Therefore, when we combine θ'' and θ' in θ , the trace slice for θ cannot belong to \mathcal{G} , due to the definition of enable set.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm $\mathbb{C}^+\langle X\rangle$ using the enable set and timestamps, as shown in Figure 16.14. This algorithm makes use of the mappings discussed above, namely, $\text{enable}_{\mathcal{G}}^X$, Δ , \mathcal{U} , disable and \mathcal{T} , and maintains an integer variable to track the timestamp. Similar to algorithm $\mathbb{C}^+\langle X\rangle$, when event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X\rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in **main**). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function `createNewMonitorStates` in algorithm $\mathbb{D}\langle X\rangle$. Unlike in algorithm $\mathbb{C}^+\langle X\rangle$, where all the parameter instances less informative than θ are searched to find all the compatible parameter instances using \mathcal{U} , `createNewMonitorStates` enumerates parameter sets in $\text{enable}_{\mathcal{G}}^X(e)$ and looks for parameter instances

Event	$e_2\langle p_2 \rangle$	$e_1\langle p_1 \rangle$	$e_3\langle p_1, p_2 \rangle$
Δ	$\langle p_2 \rangle : \sigma(i, e_2)$	$\langle p_2 \rangle : \sigma(i, e_2)$ $\langle p_1 \rangle : \sigma(i, e_1)$	$\langle p_2 \rangle : \sigma(i, e_2)$ $\langle p_1 \rangle : \sigma(i, e_1)$
\mathcal{T}	$\langle p_2 \rangle : 1$	$\langle p_2 \rangle : 1$ $\langle p_1 \rangle : 3$	$\langle p_2 \rangle : 1$ $\langle p_1 \rangle : 3$
disable	$\langle p_2 \rangle : 2$	$\langle p_2 \rangle : 2$ $\langle p_1 \rangle : 4$	$\langle p_2 \rangle : 2$ $\langle p_1 \rangle : 4$ $\langle p_1, p_2 \rangle : 5$

Figure 16.13: Another monitoring using enable sets and timestamps.

whose domains are in $\text{enable}_G^X(e)$ and which are compatible with θ , also using \mathcal{U} . The inclusion check at line 2 in `createNewMonitorStates` is to omit unnecessary search since if $\text{Dom}(\theta) \subseteq X_e$ then no new parameter instance will be created from θ . This way, `createNewMonitorStates` creates all the parameter instances that combine θ with compatible parameter instances that also satisfy the enable set of e using fewer lists in \mathcal{U} .

If e is a creation event then a monitor state for θ is initialized (line 3 in `main`). Note that $\Delta(\theta)$ can be defined in function `createNewMonitorStates` if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. `disable`(θ) is set to the current timestamp after all the creations and the timestamp is increased. The rest of function `main` in $\mathbb{D}\langle X \rangle$ is the same as in $\mathbb{C}^+\langle X \rangle$: all the relevant monitor states are updated according to e .

Function `defineNew` in $\mathbb{D}\langle X \rangle$ is similar to the one in $\mathbb{C}^+\langle X \rangle$. The only difference is that $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function `defineTo` in $\mathbb{D}\langle X \rangle$ checks `disable` and \mathcal{T} as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$.

In all of our tested cases $\mathbb{D}\langle X \rangle$ performs better than $\mathbb{C}^+\langle X \rangle$; in fact, in most cases that caused notable monitoring overhead, the efficiency of $\mathbb{D}\langle X \rangle$ is significantly better than $\mathbb{C}^+\langle X \rangle$. For example, in two extreme cases, $\mathbb{C}^+\langle X \rangle$ could not finish, while $\mathbb{D}\langle X \rangle$ had no problems. In terms of memory usage $\mathbb{D}\langle X \rangle$ also performs better, as expected, except for a few cases where $\mathbb{C}^+\langle X \rangle$ generates more garbage collections, reducing peak memory usage at the expense of performance.

```

Algorithm  $\mathbb{D}\langle X \rangle (M = (S, \mathcal{E}, \mathcal{C}, \beta, \sigma, \gamma))$ 
Input: mapping  $\text{enable}_{\mathcal{G}}^X : [\mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(X))]$ 
Globals: mapping  $\Delta : [[X \rightarrow V] \rightarrow S]$ 
         mapping  $\mathcal{T} : [[X \rightarrow V] \rightarrow \text{integer}]$ 
         mapping  $\mathcal{U} : [X \rightarrow V] \rightarrow \mathcal{P}_f([X \rightarrow V])$ 
         mapping  $\text{disable} : [[X \rightarrow V] \rightarrow \text{integer}]$ 
         integer  $\text{timestamp}$ 
Initialization:  $\mathcal{U}(\theta) \leftarrow \emptyset$  for any  $\theta$ ,  $\text{timestamp} \leftarrow 0$ 

function  $\text{main}(e\langle \theta \rangle)$ 
1  if  $\Delta(\theta)$  undefined then
2       $\text{createNewMonitorState}(e\langle \theta \rangle)$ 
3      if  $\Delta(\theta)$  undefined and  $e$  is a creation event then  $\text{defineNew}(\theta)$  endif
4       $\text{disable}(\theta) \leftarrow \text{timestamp}$ 
5       $\text{timestamp} \leftarrow \text{timestamp} + 1$ 
6  endif
7  foreach  $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$  s.t.  $\Delta(\theta')$  defined do  $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$  endfor
function  $\text{createNewMonitorStates}(e\langle \theta \rangle)$ 
1  foreach  $X_e \in \text{enable}_{\mathcal{G}}^X(e)$  (in reversed topological order) do
2      if  $\text{Dom}(\theta) \not\subseteq X_e$  then
3           $\theta_m \leftarrow \theta'$  s.t.  $\theta' \sqsubset \theta$  and  $\text{Dom}(\theta') = \text{Dom}(\theta) \cap X_e$ 
4          foreach  $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$  s.t.  $\text{Dom}(\theta'') = X_e$  do
5              if  $\Delta(\theta'')$  defined and  $\Delta(\theta'' \sqcup \theta)$  undefined then  $\text{defineTo}(\theta'' \sqcup \theta, \theta'')$  endif
6          endfor
7      endif
8  endfor
function  $\text{defineNew}(\theta)$ 
1  foreach  $\theta'' \sqsubset \theta$  do
2      if  $\Delta(\theta'')$  defined then return endif
3  endfor
4   $\Delta(\theta) \leftarrow \beta$ 
5   $\mathcal{T}(\theta) \leftarrow \text{timestamp}$ 
6   $\text{timestamp} \leftarrow \text{timestamp} + 1$ 
7  foreach  $\theta'' \sqsubset \theta$  do
8       $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$ 
9  endfor
function  $\text{defineTo}(\theta, \theta')$ 
1  foreach  $\theta'' \sqsubseteq \theta$  s.t.  $\theta'' \not\sqsubseteq \theta'$  do
2      if  $\text{disable}(\theta'') > \mathcal{T}(\theta')$  or  $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$  then return endif
3  endfor
4   $\Delta(\theta) \leftarrow \Delta(\theta')$ 
5   $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$ 
6  foreach  $\theta'' \sqsubset \theta$  do
7       $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$ 
8  endfor

```

Figure 16.14: Optimized Monitoring Algorithm $\mathbb{D}\langle X \rangle$.

16.6.2 Proofs of Correctness

The goal of this section is to show that algorithms $\mathbb{D}\langle X \rangle$ and $\mathbb{C}^+\langle X \rangle$ produce the same mapping Δ for the same given trace. $\mathbb{C}\langle X \rangle$ is already known to be correct for our definition of parametric trace monitoring due to the results in [65]. As mentioned $\mathbb{C}^+\langle X \rangle$ is a straight-forward extension of $\mathbb{C}\langle X \rangle$. Thus by showing that $\mathbb{D}\langle X \rangle$ and $\mathbb{C}^+\langle X \rangle$ produce the same Δ (i.e., showing that they behave the same), we show that $\mathbb{D}\langle X \rangle$, itself, is correct for our definition of parametric monitoring.

We fix a trace $\tau = e_1e_2\dots e_n$, a monitor $M = (S, \mathcal{E}, \mathcal{C}, \beta, \sigma, \gamma)$ and a desired value set \mathcal{G} in what follows. We use $\Delta_{\mathbb{C}}$ and $\Delta_{\mathbb{D}}$ to refer to the Δ in algorithms $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, respectively. For convenience, we also let $\text{timestamp} : [\text{integer} \rightarrow \text{integer}]$ be the function defined as follows: $\text{timestamp}(k)$ is the value of timestamp in $\mathbb{D}\langle X \rangle$ at the event e_k for $0 < k \leq n$; otherwise $\text{timestamp}(k)$ is undefined. timestamp and \mathcal{T} in $\mathbb{D}\langle X \rangle$ have the following properties:

Proposition 43 *The follow holds for timestamp and \mathcal{T} used in algorithm $\mathbb{D}\langle X \rangle$.*

1. *For $0 < k, k' \leq n$, $k \geq k'$ iff $\text{timestamp}(k) \geq \text{timestamp}(k')$.*
2. *$\Delta_{\mathbb{D}}(\theta)$ is defined iff $\mathcal{T}(\theta)$ is defined.*

Proof: 1. is obvious since timestamp is monotonic along the observed trace. 2. holds because $\Delta_{\mathbb{D}}(\theta)$ and $\mathcal{T}(\theta)$ are always defined together (lines 1 and 2 in `defineNew` and lines 6 and 7 in `defineTo`). \square

We next define two functions that describe *when* and *how* a monitor state is created for a parameter instance.

Definition 99 *Function $\text{set} : [[X \rightarrow V] \rightarrow \text{integer}]$ is defined as follows: $\text{set}(\theta) = k$ if $\Delta(\theta)$ is initialized at e_k . Function $\text{MT} : [[X \rightarrow V] \rightarrow [X \rightarrow V]^*]$ is defined as follows: $\text{MT}(\theta) = \theta_1 \dots \theta_m$ where $\theta_m = \theta$, $\Delta(\theta_1)$ is initialized to β , and $\Delta(\theta_i)$ is initialized using $\Delta(\theta_{i-1})$ at some event e for any $1 < i \leq m$.*

Intuitively, MT maps a given parameter instance to the string of parameter instances transitively used to initialize its state, this tells us the *how*. The function set clearly gives us the *when*. Obviously, for both $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, $\text{set}(\theta)$ is defined if and only if $\text{MT}(\theta)$ is defined. Let $\text{set}_{\mathbb{C}}$ and $\text{set}_{\mathbb{D}}$ be the set in algorithm $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, respectively, and let $\text{MT}_{\mathbb{C}}$ and $\text{MT}_{\mathbb{D}}$ be the MT in algorithm $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, respectively.

Proposition 44 *For algorithms $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, the following hold for set and MT:*

1. *For θ_i and θ_j in $MT(\theta)$, $\theta_i \sqsubset \theta_j$ if $i < j$.*
2. *If $MT_{\mathbb{D}}(\theta) = \theta_1 \dots \theta_m$ then $\mathcal{T}(\theta) = \text{timestamp}(\text{set}_{\mathbb{D}}(\theta_1))$.*
3. *If $\text{set}_{\mathbb{D}}(\theta)$ is defined then $\text{set}_{\mathbb{C}}(\theta)$ is defined and $\text{set}_{\mathbb{C}}(\theta) \leq \text{set}_{\mathbb{D}}(\theta)$.*
4. *If $\text{set}_{\mathbb{C}}(\theta) = \text{set}_{\mathbb{D}}(\theta)$ and $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ when they are initialized, then $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring process.*
5. *If $\text{set}_{\mathbb{C}}(\theta) = \text{set}_{\mathbb{D}}(\theta)$ and $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ then $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring process.*

Proof:

1. It follows by Definition 99 and line 6 in `createNewMonitorStates` in $\mathbb{D}\langle X \rangle$.

2. Prove by induction on the length of $MT_{\mathbb{D}}(\theta)$. If $MT_{\mathbb{D}}(\theta) = \theta$, suppose that $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k , i.e., $\text{set}_{\mathbb{D}}(\theta) = k$. Obviously, $\Delta_{\mathbb{D}}(\theta)$ is defined using `defineNew` in $\mathbb{D}\langle X \rangle$. Hence, $\mathcal{T}(\theta) = \text{timestamp}(k)$ according to line 2 in `defineNew`. Now suppose that for $0 < j$ and any θ'' s.t. $MT_{\mathbb{D}}(\theta'') = \theta_1 \dots \theta_m$ and $m < j$, $\mathcal{T}(\theta'') = \text{timestamp}(\text{set}_{\mathbb{D}}(\theta_1))$. If $MT_{\mathbb{D}}(\theta) = \theta_1 \dots \theta_j$ then $\theta = \theta_j$ and $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ by Definition 99. $\mathcal{T}(\theta_j) = \mathcal{T}(\theta_{j-1})$ according to line 7 in `defineTo` in $\mathbb{D}\langle X \rangle$. By induction, $\mathcal{T}(\theta) = \mathcal{T}(\theta_{j-1}) = \text{timestamp}(\text{set}_{\mathbb{D}}(\theta_1))$.

3. Prove by induction on the length of $MT_{\mathbb{D}}(\theta)$. We only need to show that if $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k and $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k then $\Delta_{\mathbb{C}}(\theta)$ is defined at e_k . If $MT_{\mathbb{D}}(\theta) = \theta$, suppose $\text{set}_{\mathbb{D}}(\theta) = k$ and $e_k \langle \theta' \rangle$. Since θ is not initialized with another parameter instance, it should be defined using `defineNew` function in $\mathbb{D}\langle X \rangle$, which only occurs via line 4 in `main`. Hence, $\theta' = \theta$ and e_k is a creation event. If $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k , it will be defined at e_k because line 10 in the `main` function in $\mathbb{C}^+\langle X \rangle$ will be executed if $\Delta_{\mathbb{C}}(\theta)$ is undefined before line 9.

Now suppose that for any parameter instance θ'' s.t. $\text{set}_{\mathbb{D}}(\theta'')$ is defined and the length of $MT_{\mathbb{D}}(\theta'')$ is less than j , $\text{set}_{\mathbb{C}}(\theta'') \leq \text{set}_{\mathbb{D}}(\theta'')$. If $\text{set}_{\mathbb{D}}(\theta)$ is defined and $MT_{\mathbb{D}}(\theta) = \theta_1 \dots \theta_j$ where $\theta_j = \theta$, let $\text{set}_{\mathbb{D}}(\theta) = k$ and $e_k \langle \theta' \rangle$. By Definition 99, $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Hence, $\text{set}_{\mathbb{D}}(\theta_{j-1}) < k$ and $\theta' \sqcup \theta_{j-1} = \theta$ according to line 6 in the `createNewMonitorStates` function in $\mathbb{D}\langle X \rangle$. By induction, $\text{set}_{\mathbb{C}}(\theta_{j-1}) \leq \text{set}_{\mathbb{D}}(\theta_{j-1}) < k$, that is, $\Delta_{\mathbb{C}}(\theta_{j-1})$ is defined before e_k . Therefore, if $\Delta_{\mathbb{C}}(\theta)$ is undefined before e_k , $\Delta_{\mathbb{C}}(\theta_j)$ will be

defined in $\mathbb{C}^+\langle X \rangle$ at e_k because: if $\theta' = \theta$ then $\Delta_{\mathbb{C}}(\theta)$ will be defined at line 8 in **main** in $\mathbb{C}^+\langle X \rangle$ ($\theta_{j-1} \sqsubset \theta$ by 1.); otherwise, it will be defined at line 15 in **main** ($\theta' \sqcup \theta_{j-1} = \theta$).

4. In both $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, after $\Delta(\theta)$ is defined at e_k , it will be updated using any event $e_j\langle \theta' \rangle$ with $\theta' \sqsubseteq \theta$ and $k < j$. If $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$ and $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ then $MT_{\mathbb{C}}(\theta)$ and $MT_{\mathbb{D}}(\theta)$ will be updated using the same events afterward and therefore equivalent during the whole monitoring.

5. It can be easily proved by induction on the length of $MT_{\mathbb{D}}(\theta)$ and 4. \square

The following lemma shows that $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$ are equivalent for monitors that are created from the initial state.

Lemma 2 *The following hold for MT:*

1. If $MT_{\mathbb{C}}(\theta) = \theta$ then $MT_{\mathbb{D}}(\theta) = \theta$ and $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$.
2. If $MT_{\mathbb{D}}(\theta) = \theta$ then $MT_{\mathbb{C}}(\theta) = \theta$ and $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$.

Proof:

1. If $MT_{\mathbb{C}}(\theta) = \theta$, suppose that $set_{\mathbb{C}}(\theta) = k$. Obviously, $\Delta_{\mathbb{C}}(\theta)$ is defined by the **defineNew** function in $\mathbb{C}^+\langle X \rangle$, which only occurs when e_k is a creation event and comes with the parameter instance θ . Also, for all $\theta' \sqsubset \theta$, $\Delta_{\mathbb{C}}(\theta')$ is undefined before e_k ; otherwise, $\Delta_{\mathbb{C}}(\theta)$ should be defined using $\Delta_{\mathbb{C}}(\theta')$ at line 8 in **main** in $\mathbb{C}^+\langle X \rangle$. By Proposition 44 3., $\Delta_{\mathbb{D}}(\theta)$ and $\Delta_{\mathbb{D}}(\theta')$, for all $\theta' \sqsubset \theta$, are undefined before e_k . So $\Delta_{\mathbb{D}}(\theta)$ cannot be defined in the **createNewMonitorStates** function in $\mathbb{D}\langle X \rangle$ using some $\theta' \sqsubset \theta$ when e_k is encountered. Hence, the condition at line 3 in **main** in $\mathbb{D}\langle X \rangle$ is satisfied and line 4 will be executed to initialize $\Delta_{\mathbb{D}}(\theta)$ using **defineNew** in $\mathbb{D}\langle X \rangle$. Therefore, $MT_{\mathbb{D}}(\theta) = \theta$ and $set_{\mathbb{D}}(\theta) = k = set_{\mathbb{C}}(\theta)$.

2. By Proposition 44.3., if $MT_{\mathbb{D}}(\theta) = \theta$ and $set_{\mathbb{D}}(\theta) = k$ then $MT_{\mathbb{C}}(\theta)$ is defined before or at e_k . Assume that $MT_{\mathbb{C}}(\theta) = \theta_1.. \theta_m$ and $m > 1$. Then we have 1) $\theta_1 \sqsubset \theta$ by Proposition 44 1.; 2) $MT_{\mathbb{D}}(\theta_1) = MT_{\mathbb{C}}(\theta_1) = \theta_1$ and $set_{\mathbb{C}}(\theta_1) = set_{\mathbb{D}}(\theta_1)$ 1.; 3) $set_{\mathbb{C}}(\theta_1) < set_{\mathbb{C}}(\theta) \leq set_{\mathbb{D}}(\theta)$ by Proposition 44.3. Let $e_k\langle \theta' \rangle$. Since $MT_{\mathbb{D}}(\theta) = \theta$, $\Delta_{\mathbb{D}}(\theta)$ is defined using **defineNew** via line 4 in **main** in $\mathbb{D}\langle X \rangle$ when e_k is encountered. Hence, $\theta = \theta'$. However, since $\Delta_{\mathbb{D}}(\theta_1)$ is defined before e_k , the condition at line 2 in **defineNew** is satisfied and $\Delta_{\mathbb{D}}(\theta)$ cannot be defined at e_k . Contradiction reached. Therefore, $MT_{\mathbb{C}}(\theta) = \theta$. By 1., $set_{\mathbb{C}}(\theta) = set_{\mathbb{D}}(\theta)$. \square

Proposition 45 *For algorithms $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$, the following hold:*

1. If $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ then for any $\theta' \in MT_{\mathbb{C}}(\theta)$, $set_{\mathbb{C}}(\theta') = set_{\mathbb{D}}(\theta')$.
2. If $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$ then $\Delta_{\mathbb{C}}(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring.

Proof:

1. Suppose $MT_{\mathbb{C}}(\theta) = \theta_1, \dots, \theta_m$. Prove by induction on $MT_{\mathbb{C}}(\theta)$. For θ_1 , since $MT_{\mathbb{C}}(\theta_1) = \theta_1$, $set_{\mathbb{C}}(\theta_1) = set_{\mathbb{D}}(\theta_1)$ by Lemma 2.1. Now suppose that for some $1 < j \leq m$, $set_{\mathbb{C}}(\theta_i) = set_{\mathbb{D}}(\theta_i)$ for any $0 < i < j$. Assume that $set_{\mathbb{C}}(\theta_j) \neq set_{\mathbb{D}}(\theta_j)$. We have $set_{\mathbb{C}}(\theta_j) < set_{\mathbb{D}}(\theta_j)$ by Proposition 44.3. Let $set_{\mathbb{C}}(\theta_j) = k$ and $e_k \langle \theta'' \rangle$. Since $\theta'' \sqcup \theta_{j-1} = \theta_j$, we have $\theta'' \not\sqsubseteq \theta_{j-1}$. Also, $disable(\theta'') > timestamp(k) > \mathcal{T}(\theta_{j-1})$ after e_k . Let $set_{\mathbb{D}}(\theta) = g$. We have that $\Delta_{\mathbb{D}}(\theta_j)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{j-1})$ because $g > k$ and θ'' will satisfy the condition at line 2 in `defineTo` in $\mathbb{D}\langle X \rangle$. Contradiction found. Therefore, $set_{\mathbb{C}}(\theta_j) = set_{\mathbb{D}}(\theta_j)$.

2. Follow by 1. and Proposition 44.5. □

Let $\Delta_{\mathbb{C}}^{\tau}$ be the Δ after $\mathbb{C}^+\langle X \rangle$ processes τ and $\Delta_{\mathbb{D}}^{\tau}$ be the Δ after $\mathbb{D}\langle X \rangle$ processes τ .

Proposition 46 *The following holds:*

1. If $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) \in \mathcal{G}$ and for any $\theta_i \in MT_{\mathbb{C}}(\theta)$, $i > 1$, let $set_{\mathbb{C}}(\theta_i) = k$, we have $Dom(\theta_{i-1}) \in enable_{\mathcal{G}}^X(e_k)$.
2. If $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) \in \mathcal{G}$ then $MT_{\mathbb{C}}(\theta) = MT_{\mathbb{D}}(\theta)$.

Proof:

1. Suppose that the sliced trace for θ is $\tau_{\theta} = e'_1 \langle \theta'_1 \rangle \dots e'_h \langle \theta'_h \rangle$. Then $\sigma(\tau_{\theta}) = \Delta_{\mathbb{C}}^{\tau}(\theta)$, according to Theorem 3 in [65]. Since $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) \in \mathcal{G}$, $P(\tau_{\theta}) \in \mathcal{G}$. Also, since $\Delta_{\mathbb{C}}(\theta_i)$ is defined at e_k , $e_k \in \tau_{\theta}$ and it is the first occurrence of e_k in τ_{θ} . Suppose that e'_n is the first occurrence of e_k in τ_{θ} . Then $enable_{\tau}(e_k) = \{e'_1, \dots, e'_{n-1}\}$ by Definition 96. For any $0 < j < n$, let $e'_j \langle \theta'' \rangle$, then $\theta'' \sqsubseteq \theta_{i-1}$; otherwise, e'_j should not be contained in the slice for θ_{i-1} and thus not in the slice for θ_i (since $\Delta_{\mathbb{C}}(\theta_i)$ is initialized using $\Delta_{\mathbb{C}}(\theta_{i-1})$.) Hence, $\cup_{\{e'_1, \dots, e'_{n-1}\}}^X = Dom(\theta_{i-1})$, that is, $Dom(\theta_{i-1}) \in enable_{\mathcal{G}}^X(e_k)$ by Definition 98.

2. Suppose that $MT_{\mathbb{C}}(\theta) = \theta_1, \dots, \theta_m$. Prove by induction on $MT_{\mathbb{C}}(\theta)$. For θ_1 , $MT_{\mathbb{C}}(\theta_1) = \theta_1$. Hence, $MT_{\mathbb{D}}(\theta_1) = \theta_1$ by Lemma 2. Now suppose that for some $1 < j \leq m$, we have $MT_{\mathbb{D}}(\theta_{j-1}) = MT_{\mathbb{C}}(\theta_{j-1}) = \theta_1, \dots, \theta_{j-1}$. Let

$set_{\mathbb{C}}(\theta_j) = k$ and $e_k\langle\theta'\rangle$. By Proposition 44.3., $\Delta_{\mathbb{D}}(\theta_j)$ is undefined before e_k . Also, $\theta' \sqcup \theta_{j-1} = \theta_j$ due to line 15 in `main` in $\mathbb{C}^+\langle X \rangle$.

By 1., $\text{Dom}(\theta_{j-1}) \in \text{enable}_{\mathcal{G}}^X(e_k)$. Hence, $\Delta_{\mathbb{D}}(\theta_j)$ will be defined at e_k because of the loop from line 4 - 8 in `createNewMonitorStates` in $\mathbb{D}\langle X \rangle$. We only need to show that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Assume that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta'')$ and $\theta'' \neq \theta_{j-1}$. Then we have $\theta'' \sqcup \theta' = \theta_j$. $\theta'' \not\sqsubseteq \theta_{j-1}$ because the loop from line 1 to line 10 in `createNewMonitorStates` in $\mathbb{D}\langle X \rangle$ is carried out in a reverse topological order. Also, $\theta_{j-1} \not\sqsubseteq \theta''$ because the loops from line 2 to line 6 and from line 12 to line 18 in `main` in $\mathbb{C}^+\langle X \rangle$ are carried out in a reverse topological order. Such situation, i.e., θ_j does not have a maximum sub-instance, is impossible according to the proof for algorithm $\mathbb{A}\langle X \rangle$ in [65]. Contradiction found. Therefore, $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ at e_k . We then have $\text{MT}_{\mathbb{D}}(\theta_j) = \text{MT}_{\mathbb{D}}(\theta_{j-1})\theta_j = \text{MT}_{\mathbb{C}}(\theta_{j-1})\theta_j = \text{MT}_{\mathbb{C}}(\theta_j)$. By induction, $\text{MT}_{\mathbb{C}}(\theta_m) = \text{MT}_{\mathbb{D}}(\theta_m)$. \square

Proposition 47 *If $\Delta_{\mathbb{D}}^T(\theta)$ is defined then $\text{MT}_{\mathbb{C}}(\theta) = \text{MT}_{\mathbb{D}}(\theta)$.*

Proof: Suppose that $\text{MT}_{\mathbb{D}}(\theta) = \theta_1, \dots, \theta_m$. Prove by induction on $\text{MT}_{\mathbb{D}}(\theta)$. For θ_1 , $\text{MT}_{\mathbb{D}}(\theta_1) = \theta_1$. Hence, $\text{MT}_{\mathbb{C}}(\theta_1) = \theta_1$ by Lemma 2.2. Now suppose that for some $1 < j \leq m$, we have $\text{MT}_{\mathbb{D}}(\theta_{j-1}) = \text{MT}_{\mathbb{C}}(\theta_{j-1}) = \theta_1, \dots, \theta_{j-1}$. Let $set_{\mathbb{D}}(\theta_j) = k$ and $e_k\langle\theta'\rangle$.

Suppose that $\text{MT}_{\mathbb{C}}(\theta_j) = \theta_1^j \dots \theta_h^j$ where $\theta_h^j = \theta_j$. We first show that $\theta_1 = \theta_1^j$ by contradiction. Assume $\theta_1 \neq \theta_1^j$. Let $set_{\mathbb{C}}(\theta_1^j) = p^j$ and $set_{\mathbb{D}}(\theta_1) = p$. Since $\text{MT}_{\mathbb{C}}(\theta_1^j) = \theta_1^j$ and $\text{MT}_{\mathbb{D}}(\theta_1) = \theta_1$, we have that $e_{p^j}\langle\theta_1^j\rangle, e_p\langle\theta_1\rangle$ and they are both creation events. We also have $\mathcal{T}_{\mathbb{D}}(\theta_1) = \text{timestamp}(p)$. By Proposition 44.2, $\Delta_{\mathbb{D}}(\theta_1^j)$ is not defined before p^j . Hence, $\Delta_{\mathbb{D}}(\theta_1^j)$ is defined at p^j and $\mathcal{T}_{\mathbb{D}}(\theta_1^j) = \text{timestamp}(p^j)$. Also, $\text{disable}(\theta_1^j) > \mathcal{T}_{\mathbb{D}}(\theta_1^j)$ since line 6 in `main` of algorithm $\mathbb{D}\langle X \rangle$ is executed after $\mathcal{T}_{\mathbb{D}}(\theta_1^j)$ is defined at line 4. Since $\theta_1 \neq \theta_1^j$, $p^j \neq p$; in other words, either $p^j < p$ or $p^j > p$. Therefore, either $\mathcal{T}_{\mathbb{D}}(\theta_1^j) < \mathcal{T}_{\mathbb{D}}(\theta_1)$ or $\mathcal{T}_{\mathbb{D}}(\theta_1) < \mathcal{T}_{\mathbb{D}}(\theta_1^j) < \text{disable}(\theta_1^j)$ by Proposition 43.1. Let θ_n be the first parameter instance in $\text{MT}_{\mathbb{D}}(\theta_j)$ s.t. $\theta_1^j \sqsubset \theta_n$ and $\theta_1^j \not\sqsubseteq \theta_{n-1}$, $n > 1$, and let $set_{\mathbb{D}}(\theta_n) = p_n$. Then $\Delta_{\mathbb{D}}(\theta_n)$ is defined in the `defineTo` function in $\mathbb{D}\langle X \rangle$ at e_{p_n} using $\Delta_{\mathbb{D}}(\theta_{n-1})$. However, it is impossible since θ_1^j satisfies the condition at line 2 in `defineTo` and prevents defining $\Delta_{\mathbb{D}}(\theta_n)$ at e_{p_n} . Contradiction found and $\theta_1 = \theta_1^j$.

Assume that $\text{MT}_{\mathbb{C}}(\theta_j) \neq \text{MT}_{\mathbb{D}}(\theta_j)$. We can find $l > 1$ s.t. $\theta_l^j \neq \theta_l$ and $\theta_i^j = \theta_i$ for any $0 < i < l$. Let $set_{\mathbb{C}}(\theta_l^j) = k$ and $set_{\mathbb{C}}(\theta_l) = g$. Suppose $e_{n_l}\langle\theta''\rangle$.

We have $\theta_{l-1}^j \sqcup \theta'' = \theta_l^j$; so $\theta'' \not\sqsubseteq \theta_{l-1}^j$. Also, $\text{disable}(\theta'') > \mathcal{T}(\theta_l^j) = \mathcal{T}(\theta_1^j) = \mathcal{T}(\theta_1)$ after e_k . $k < g$ is impossible; otherwise, $\Delta_{\mathbb{D}}(\theta_l)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{l-1})$ because θ'' will satisfy the condition at line 2 in **defineTo** in $\mathbb{D}\langle X \rangle$. Hence, $k > g \geq \text{set}_{\mathbb{C}}(\theta_l)$ by Proposition 44.3. In other words, $\Delta_{\mathbb{C}}(\theta_l)$ is defined before e_k . Therefore, $\theta_l \notin \text{MT}_{\mathbb{C}}(\theta_j)$ but $\theta_l \subseteq \theta_j$. Then we can find $\theta_p^j \in \text{MT}_{\mathbb{C}}(\theta_j)$ s.t. $\theta_l \sqsubset \theta_p^j$ and $\theta_l \not\sqsubseteq \theta_i$ for any $0 < i < p$. However, suppose $\text{set}_{\mathbb{C}}(\theta_p^j) = n$, then at event e_n , we have $\theta_l \sqsubset \theta_p^j$ and $\theta_l \not\sqsubseteq \theta_{p-1}^j$. According to the proof for algorithm $\mathbb{A}\langle X \rangle$ in [65], we should have $\theta_{p-1}^j \sqsubset \theta_l$, which means that $\Delta_{\mathbb{C}}(\theta_p^j)$ should be defined using $\Delta_{\mathbb{C}}(\theta_l)$. Contradiction found. Therefore, $\text{MT}_{\mathbb{C}}(\theta_j) = \text{MT}_{\mathbb{D}}(\theta_j)$. \square

Theorem 26 *The following holds:*

1. if $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) \in \mathcal{G}$ then $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}(\theta))$;
2. if $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) \in \mathcal{G}$ then $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{D}}^{\tau}(\theta))$;
3. $\gamma(\Delta_{\mathbb{C}}^{\tau}(\theta)) \in \mathcal{G}$ iff $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}(\theta))$ iff $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) \in \mathcal{G}$.

Proof:

1. By Proposition 46 and Proposition 45.2, $\Delta_{\mathbb{D}}^{\tau}(\theta) = \Delta_{\mathbb{C}}^{\tau}(\theta)$. Hence, $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}(\theta))$.

2. Follow by Proposition 47 and Proposition 45.2.

3. Follow by 1 and 2. \square

Theorem 26 states that a trace slice for θ is reported by $\mathbb{C}^+\langle X \rangle$ to be in \mathcal{G} if and only if it is also reported by $\mathbb{D}\langle X \rangle$ to be in \mathcal{G} . In other words, $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$ are equivalent for those parameter instances whose trace slices are in \mathcal{G} . Thus $\mathbb{D}\langle X \rangle$ is complete and sound.

16.7 Implementation and Evaluation

We implemented code generation for Algorithms $\mathbb{C}^+\langle X \rangle$ and $\mathbb{D}\langle X \rangle$ in Java-MOP. The indexing technique proposed in [62] is used to implement all the mappings in the algorithms. Some optimizations were applied to $\mathbb{D}\langle X \rangle$ to achieve more efficient monitoring code. First, a method is generated for each event and in the method, $\text{enable}_{\mathcal{G}}^X(e)$ is statically determined. Thus the main loop in `createNewMonitorStates` is unrolled in the monitoring code.

In every iteration of the unrolled loop, X_e is statically determined. Hence, the condition at line 2 and θ_m at line 3 in `createNewMonitorStates` can be statically computed for each iteration and the resulting values are constants in the generated code. The invocation of function `defineTo` at line 6 in `createNewMonitorStates` is statically expanded using the function body of `defineTo` in every unrolled iteration of the main loop. This way, the context information of call sites can be used to optimize every copy of the `defineTo` function. For example, the domains of θ and θ' are fixed in each iteration of the unrolled loop in `createNewMonitorStates`, so we can also unroll the loop from line 1 to line 5 in `defineTo` and compute the comparison between θ' and θ'' at code generation time. Also, the inner loop (lines 4 - 8) in `createNewMonitorStates` checks every parameter instance in $\mathcal{U}(\theta)$ but $\mathcal{U}(\theta)$ may contain many other instances whose domains are not X_e . To reduce runtime overhead, the code generation makes a mapping for each e and $X_e \in \text{enable}_G^X(e)$. Specifically, given an event definition \mathcal{D}_E , for any event e and every $X_e \in \text{enable}_G^X(e)$, a mapping $\mathcal{U}_{X_e}^e$ is generated to map the parameter instance θ_m with $\text{Dom}(\theta_m) = \mathcal{D}_E(\theta) \cap X_e$ to a list of parameter instances more informative than θ_m whose domain is X_e . In every iteration of the unrolled loop in `createNewMonitorStates`, the corresponding $\mathcal{U}_{X_e}^e$ is used for the inner loop. This way, fewer parameter instances are enumerated at runtime.

We evaluated $\mathbb{C}^+\langle X \rangle$, $\mathbb{D}\langle X \rangle$, and Tracematches on the DaCapo benchmark suite [36]. We omitted other runtime systems because they have been evaluated and compared with either Tracematches or the original JavaMOP algorithm⁶ in other papers [9, 25, 62]. Note that Soot [253], the underlying bytecode engine for Tracematches, cannot handle the DaCapo benchmark properly, resulting in fewer instrumentation points in the `pmd` program. Accordingly, we modify our specification to have the same scope of instrumentation for a fair comparison. The raw results can be found at [94].

Experimental Settings. Our experiments were performed on a machine with 2GB RAM and a Pentium 4 2.66GHz processor. The machine’s operating system is Ubuntu Linux 7.10, and we used version 2006-10 of the DaCapo benchmark suite [36]. The default input for DaCapo was used, and we use the `-converge` option to ensure the validity of our test by running each

⁶The original JavaMOP is conservatively extended by the new $\mathbb{D}\langle X \rangle$, in that for properties supported by the original JavaMOP algorithm, $\mathbb{D}\langle X \rangle$ generates the same monitoring code and instrumentation.

	UnsafeMapIterator			SafeSyncCollection			SafeSyncMap		
	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$
antlr	-2	5	2	-2	2	1	-3	2	1
bloat	>10000	OOM	935	1448	735	712	2267	858	660
chart	-1	4	0	0	1	1	1	3	0
eclipse	8	2	1	0	0	0	0	1	1
fop	11	-2	-3	-4	-3	0	16	-5	-3
hsqldb	29	0	0	24	0	0	22	-1	0
kython	57	11	7	6	-4	-4	8	-4	-5
luindex	7	12	5	0	1	1	3	1	4
lusearch	9	1	-1	9	1	1	8	2	-1
pmd	>10000	OOM	196	33	18	15	50	21	12
xalan	10	4	4	7	-1	1	6	0	0

	SafeIterator			SafeFile			SafeFileWriter		
	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$	TM	$\mathbb{C}^+\langle X \rangle$	$\mathbb{D}\langle X \rangle$
antlr	0	0	0	N/A	11	9	N/A	2	5
bloat	11258	769	749	N/A	3	1	N/A	5	0
chart	11	5	3	N/A	-1	0	N/A	-1	0
eclipse	2	0	1	N/A	2	2	N/A	1	2
fop	5	4	1	N/A	-3	-3	N/A	-3	-5
hsqldb	17	-1	0	N/A	2	0	N/A	0	-1
kython	16	-2	0	N/A	-4	-4	N/A	-3	-5
luindex	9	3	5	N/A	22	21	N/A	-1	-1
lusearch	34	4	2	N/A	0	-1	N/A	-1	0
pmd	196	19	14	N/A	-2	0	N/A	-2	-2
xalan	10	9	8	N/A	0	-1	N/A	2	1

Figure 16.15: Average *percent* runtime overhead for Tracematches(TM), $\mathbb{C}^+\langle X \rangle$, and $\mathbb{D}\langle X \rangle$ (convergence within 3%, OOM = Out of Memory). $\mathbb{D}\langle X \rangle$, at its worst, has less than an order of magnitude of overhead.

test multiple times, until the execution time converges. After convergence, the runtime is stabilized within 3%, thus numbers in Figure 16.15 should be interpreted as “ $\pm 3\%$ ”. Additional code introduced by the AspectJ weaving process changes the program structure in DaCapo; sometimes this causes the benchmark to run a little bit faster due to better instruction cache layout.

Properties. We used the following properties in our experiments. They were borrowed from [40, 41, 195].

- **UnsafeMapIterator:** Do not update a **Map** when using the **Iterator** interface to iterate its values or its keys;
- **SafeSyncCollection:** If a **Collection** is synchronized, then its iterator also should be accessed in a synchronized manner;
- **SafeSyncMap:** If a **Collection** is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner;
- **SafeIterator:** Do not update a **Collection** when using the **Iterator** interface to iterate its elements;
- **SafeFile:** All file opens should be closed strictly in the function where it is opened;
- **SafeFileWriter:** No write to a **FileWriter** after closing.

UnsafeMapIterator, **SafeSyncCollection**, **SafeSyncMap** and **SafeFile** could not be monitored using the original JavaMOP algorithm, as they contain creation events that do not instantiate all the parameters. **SafeFile** and **SafeFileWriter** cannot be expressed in Tracematches because they are context-free properties. We use them to demonstrate the effectiveness of the enable set optimization on CFG properties. **SafeIterator** was chosen because it has generated some of the largest runtime overheads in previous experiments [62, 195].

Results and Discussions. Figure 16.15 summarizes the results of our experiments. It shows the percent overheads of $\mathbb{C}^+\langle X \rangle$, $\mathbb{D}\langle X \rangle$ (both implemented in JavaMOP), and Tracematches. All the properties were heavily monitored in the experiments. Millions of parameter instances were observed for some properties under monitoring, e.g., **SafeIterator**, putting a critical test on the generated monitoring code. All three systems generated low runtime

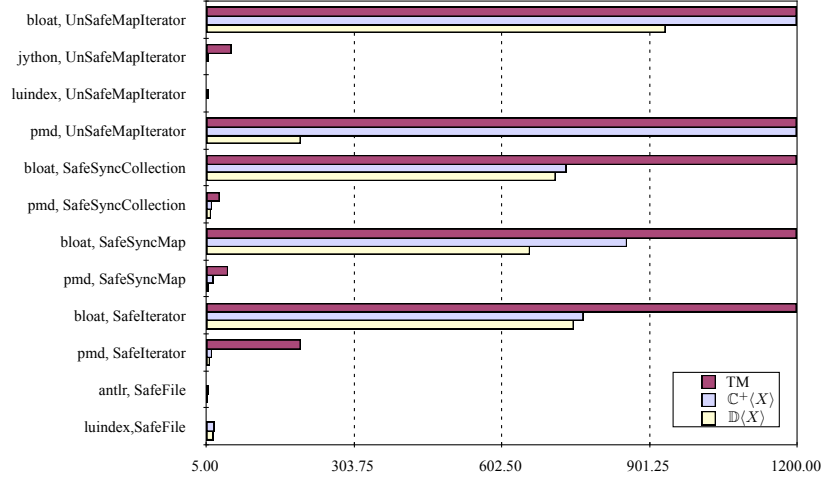


Figure 16.16: Runtime overhead statistics.

overhead in most experiments, showing their efficiency. For $\mathbb{D}(X)$, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for $\mathbb{C}^+(X)$ and Tracematches are 9 out of 66 and 15 out of 44, respectively. Figure 16.16 shows the comparison among three systems using the cases where at least two of them generated more than 10% overhead. In all cases, $\mathbb{D}(X)$ outperformed the other two and $\mathbb{C}^+(X)$ is better than Tracematches. This shows that JavaMOP provides an efficient solution to monitor parametric specifications despite its genericity in terms of specification formalisms.

The results also illustrate the effectiveness of the proposed optimization based on the enable set: on average, the overhead of $\mathbb{D}(X)$ is about 20% less than $\mathbb{C}^+(X)$. Moreover, when the property to monitor becomes more complicated, the improvement achieved by the optimization is more significant. In the two extreme cases, namely, **bloat-UnsafeMapIterator** and **pmd-UnsafeMapIterator**, where both the non-optimized JavaMOP and Tracematches crashed, the optimized JavaMOP managed to finish the executions with overheads that are reasonable for many applications, such as testing and debugging.

Figure 16.17 shows the maximum memory usages of our experiments in \log_{10} scale, in Megabytes. Only the cases where significant overhead was incurred are shown.

Two interesting observations can be made from Figure 16.17. First, the enable set optimization does not always reduce *peak* memory usage. In 7 out of

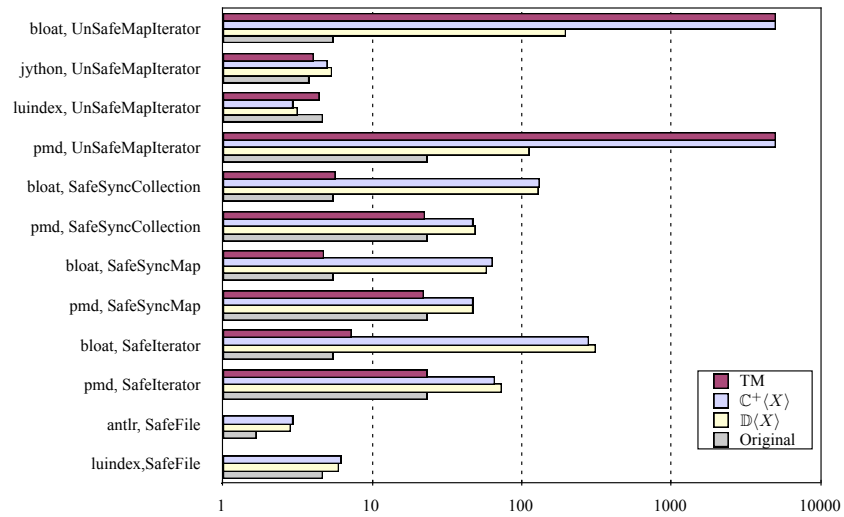


Figure 16.17: Peak memory usage statistics.

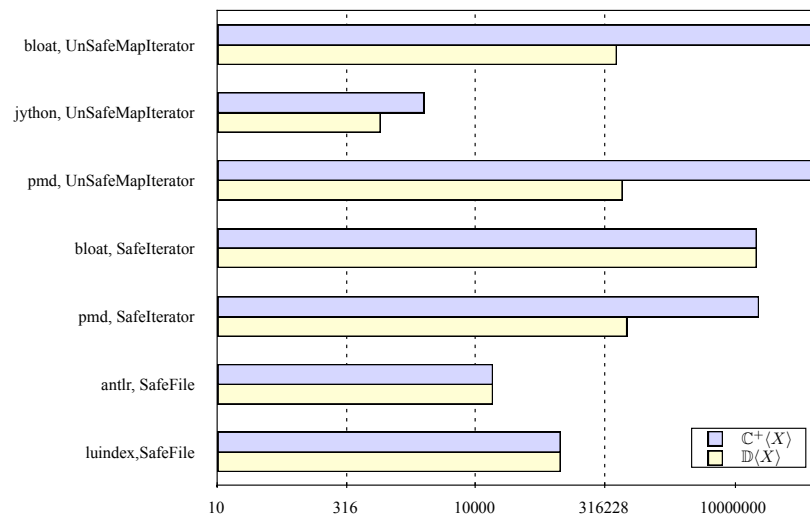


Figure 16.18: Number of parameter instance monitors.

12 cases, the $\mathbb{D}\langle X \rangle$ has lower peak memory usage than $\mathbb{C}^+\langle X \rangle$. As mentioned, **bloat-SafeMapIterator** and **pmd-SafeMapIterator**, where $\mathbb{C}^+\langle X \rangle$ did not finish execution, exiting with an out of memory error, $\mathbb{D}\langle X \rangle$ managed to complete. In the other 5 cases, $\mathbb{C}^+\langle X \rangle$ has slightly lower peak memory usage than $\mathbb{D}\langle X \rangle$. This is because $\mathbb{C}^+\langle X \rangle$ caused more garbage collections than $\mathbb{D}\langle X \rangle$. For example, in **pmd-SafeSyncCollection**, the $\mathbb{C}^+\langle X \rangle$ had 1791 young generation garbage collections while $\mathbb{D}\langle X \rangle$ had 1465 collections. However, fewer garbage collection cycles contribute to the performance increase of the enable set optimization. The reduced memory usage of **luindex-UnsafeMapIterator** in comparison to the base program can also be attributed to an increase in the number of garbage collections, 3857 for the base program, and 3957 and 4260 for the optimized and non-optimized JavaMOP runs, respectively. Another observation is that memory usage is *not directly related* to the resulting monitoring overhead: little memory overhead was observed for **bloat-SafeSyncCollection** and **bloat-SafeSyncMap**, which caused huge runtime overhead, while for **pmd-SafeIterator**, we observed high memory usage but insignificant runtime overhead.

Figure 16.18 shows the number of created monitor instances, another important measurement for our approach, also in \log_{10} scale. Only 7 cases are shown because others generated many fewer monitor instances. For the experiments with significant runtime overhead. In **bloat-SafeMapIterator** and **pmd-SafeMapIterator**, the number of instance monitors in $\mathbb{C}^+\langle X \rangle$ is not precise, due to the out of memory crash. In all cases, the optimized JavaMOP generated an equal or lesser number of monitors than the non-optimized one, showing that the optimization is effective in reducing the number of monitors, particularly in the cases where many instance monitors are created. Also, the results indicate that the number of created monitor instances is not the only factor influencing runtime overhead: no monitor instances were created for **bloat-SafeSyncCollection** and **bloat-SafeSyncMap**, but they generated significant monitoring overhead. A further inspection revealed that in both cases, a tremendous number of related events were observed: 137880368 and 165269166 events for **bloat-SafeSyncCollection** and **bloat-SafeSyncMap**, respectively. This resulted in intensive monitoring work even when no monitor instances were created. Static *program* analysis may provide a better solution in such cases, which we plan to explore in our future work.

16.8 Conclusion

Efficient monitoring of parametric properties is a very challenging problem, due to the potentially huge number of parameter instances. Until now, solutions to this problem have either used a hardwired logical formalism, or limited their handling of parameters. Our approach, based on a general semantics of parametric traces with a property-based optimization, overcomes these limitations, without sacrificing any efficiency, as our evaluation shows.

Chapter 17

Garbage Collection for Monitoring Parametric Properties

Material from [160]

Abstract: Parametric properties are behavioral properties over program events that depend on one or more parameters. Parameters are bound to concrete data or objects at runtime, which makes parametric properties particularly suitable for stating multi-object relationships or protocols. Monitoring parametric properties independently of the employed formalism involves *slicing* traces with respect to *parameter instances* and sending these slices to appropriate non-parametric *monitor instances*. The number of such instances is theoretically unbounded and tends to be enormous in practice, to an extent that how to efficiently manage monitor instances has become one of the most challenging problems in runtime verification. The previous formalism-independent approach was only able to do the obvious, namely to garbage collect monitor instances when all bound parameter objects were garbage collected. This led to pathological behaviors where unnecessary monitor instances were kept for the entire length of a program. This paper proposes a new approach to garbage collecting monitor instances. Unnecessary monitor instances are collected lazily to avoid creating undue overhead. This lazy collection, along with some careful engineering, has resulted in RV, the most efficient parametric monitoring system to date. Our evaluation shows that the average overhead of RV in the DaCapo benchmark is 15%,

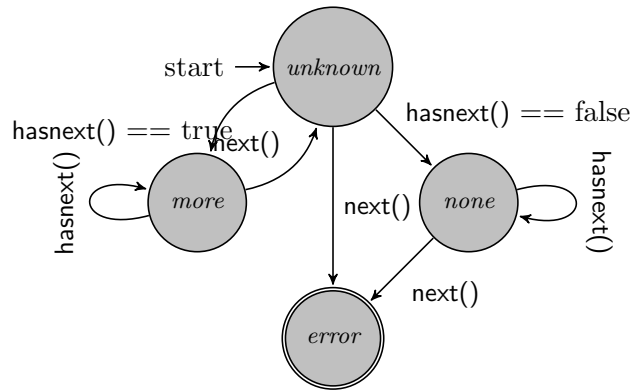


Figure 17.1: Typestate description for HASNEXT

which is two times lower than that of JavaMOP and orders of magnitude lower than that of Tracematches.

17.1 Introduction

Monitoring is an effective technique for ensuring software reliability. The well known concept of *typestate* [259] property can be enforced by using monitoring techniques. Typestates refine the notion of type by stating not only what operations are allowed by a particular object, but also what operations are allowed in what contexts.

Figure 17.1 shows the typestate description for HASNEXT. The HASNEXT typestate says that it is invalid to call the `next()` method on an `Iterator` object when there are no more elements in the underlying `Collection`, i.e., when `hasnext()` returns false, or when it is unknown if there are more elements in the `Collection`, i.e., `hasnext()` is not called. From the *unknown* state, it is always an error to call the `next()` method because such an operation could be unsafe. If `hasnext()` is called and returns true, it is safe to call `next()`, so the typestate enters the *more* state. If, however, the `hasnext()` method returns false, there are no more elements, and the typestate enters the *none* state. In the *more* and *none* states, calling the `hasnext()` method provides no new information. It is safe to call `next()` from the *more* state, but it becomes unknown if more elements exist, so the typestate reenters the initial *unknown* state. Finally, calling `next()` from the *none* state results in an error.

Typestate Property Example- It is straightforward to encode this, and all typestate properties, as particular (one-parameter) *parametric* properties. Figure 17.2 shows this property using the RV system presented in this paper. For demonstration purposes, we specify the same property using two different formalisms. The first formalism is a direct translation of the typestate diagram using the finite state machine (FSM) capabilities of RV, and is denoted by the `fsm` keyword. Each state in the typestate is written as a name followed by its transitions (i.e., monitored events) in brackets. The first state in the FSM description (`unknown`) is always considered to be the initial state of a finite state machine in the RV system. The second formalism is linear temporal logic (LTL), prefixed by the keyword `ltl`. The LTL formula here states that any call to `next()` must *always* (`[]`) be immediately preceded (`(*)`) by a call to `hasnext()` that returned true. The RV system supports multiple logical formalisms, and as this example demonstrates, some formalisms can lead to much more succinct and easy to understand specifications.

The monitored events are prefixed by the keyword `event` and are encoded using slightly extended AspectJ [164] pointcuts. One such extension is the `condition` pointcut, which is similar to the `if` pointcut: it ensures that the given pointcut is only applied if its condition is true, but unlike the `if` pointcut, it is able to refer to variables bound by returning advice. Thus, the `hasnexttrue` event is only generated if `hasnext()` returns true, and the `hasnextfalse` is only generated if `hasnext` returns false. The per `Iterator` nature of the `HASNEXT` typestate is encoded by using *one* `Iterator` parameter.

For each property, the code block following it is referred to as a *handler*. The handler is executed when a condition of its corresponding specification is met. For instance, the FSM handler in Figure 17.2 is executed when the machine enters the *error* state, while the LTL handler is executed when the LTL formula is violated. Handlers may contain any arbitrary Java code, but here they simply print messages. Such behavior is useful for debugging and testing purposes.

Parametric properties properly *generalize* typestates, as we shall see, by allowing more parameters. This allows us to specify not only properties about a given object such as the `HASNEXT` example, but also properties that specify *relationships between objects*.

General Parametric Property Examples- Figure 17.3 shows a property for the unsafe use of `Collection` and `Iterator`. The property flags it as an error if an `Iterator` is created, its underlying `Collection` is modified, and then the `Iterator` is used again. Here we use the extended regular

```

HasNext(Iterator i) {
  event hasnexttrue after(Iterator i) returning(boolean b) :
    call(* Iterator.hasNext()) && target(i) && condition(b) {}
  event hasnextfalse after(Iterator i) returning(boolean b) :
    call(* Iterator.hasNext()) && target(i) && condition(!b) {}
  event next before(Iterator i) :
    call(* Iterator.next()) && target(i) {}

  fsm :
    unknown [
      hasnexttrue -> more
      hasnextfalse -> none
      next -> error
    ]
    more [
      hasnexttrue -> more
      next -> unknown
    ]
    none [
      hasnextfalse -> none
      next -> error
    ]
    error [ ]
  @error {
    System.out.println("improper Iterator use found!");
  }

  ltl: [] (next => (*)hasnexttrue)
  @violation {
    System.out.println("improper Iterator use found!");
  }
}

```

Figure 17.2: HASNEXT property in RV using FSM and LTL

```

UnsafeIter(Collection c, Iterator i) {
  event create after(Collection c) returning(Iterator i) :
    call(Iterator Collection.iterator()) && target(c) {}
  event update after(Collection c) :
    (call(* Collection.remove*(..))
    || call(* Collection.add*(..))
    || call(* Collection.clear*(..))) && target(c){}
  event next before(Iterator i) :
    call(* Iterator.next()) && target(i){}

  ere : update* create next* update+ next
  @match {
    System.out.println("improper Concurrent Modification found!");
  }
}

```

Figure 17.3: UNSAFEITER property in RV using the ERE plugin

expression (ERE) capabilities of the RV system, as specified by the `ere` keyword. The occurrence of `update*` at the beginning of the pattern allows any number of updates before the first `create` event, in which the `Iterator` is first created. We wish to catch this behavior because Java `Collections` do not allow concurrent modification. The JVM usually throws a runtime exception when this occurs, but the exception is not guaranteed to be thrown in a multi-threaded environment.

Figure 17.4 shows a specification for the safe use of reentrant locks, called `SAFELock`, stating that the number of calls of an `acquire()` in a given method is balanced with the number of calls to `release()`. This property is parametric both in the `Lock` in question and in the `Thread`, and is specified using the context-free grammar (CFG) plugin of the RV system. The events `begin` and `end` refer to the beginning and end of *every* method. The `thread` pointcut is also an RV extension of standard AspectJ pointcuts that allows for binding the current `Thread` of execution in the monitored program. The pattern for the specification, prefixed by the keyword `cfg`, has `S` for its start symbol. The first symbol seen is always assumed the start symbol. The CFG pattern requires that `begin` and `end` events are matched and properly nested with `acquire` and `release` events, which must also be matched.

Monitoring parametric properties in their full generality is a complex task. Several parametric monitoring systems such as `Eagle` [83], `J-Lo` [258, 38, 43], `Tracematches` [9, 25], `JavaMOP` [59, 62], `PTQL` [118], `PQL` [192], `QVM` [16], `SpoX` [122], `PoET` [95], and `RuleR` [31] have been proposed in recent years. In parametric monitoring systems, the parameters are dynamically bound to objects at runtime, thus resulting in a potentially unlimited number of

```

SafeLock(Lock l, Thread t){
  event acquire before(Lock l, Thread t):
    call(* Lock.acquire()) && target(l) && thread(t) {}
  event release before(Lock l, Thread t):
    call(* Lock.release()) && target(l) && thread(t) {}
  event begin before(Thread t) :
    execution(* *.*(..) && thread(t) && !within(Lock+) {})
  event end after(Thread t) :
    execution(* *.*(..) && thread(t) && !within(Lock+) {})

  cfg : S -> S begin S end | S acquire S release | epsilon
  @fail {      System.out.println("improper Lock use found!");  }
}

```

Figure 17.4: SAFELOCK property in RV using the CFG plugin

monitor instances, one per combination of parameter bindings. The main challenge underlying the monitoring of parametric properties is therefore how to effectively manage these monitor instances, in particular how to efficiently retrieve all the monitor instances interested in an event when it takes place, and how to efficiently garbage collect monitor instances which have become unnecessary.

Earlier attempts such as Tracematches [9, 25] are careful to manage their memory, but hardwire their property specification formalism (regular expression only). JavaMOP [62, 59] is generic in specification formalisms, but, however, it has memory leaks. Due to JavaMOP’s creation of separate monitor instances in order to handle each separate parameters instantiation, recognizing and removing unnecessary monitor instances is quite challenging. JavaMOP is only able to collect a monitor instance when all the bound parameters are garbage collected, which ensures that no event can happen to the corresponding monitor instance. The problem with this method of garbage collection can be clearly seen in the UNSAFEITER property from Figure 17.3. Because it is the next event at the end of the pattern that actually causes the error, there is no way to ever match the pattern if the `Iterator` bound to a given monitor instance is garbage collected. However, JavaMOP is only able to collect the associated monitor instance if *both* the `Collection` and the `Iterator` are garbage collected. Unfortunately, in most realistic programs, `Collections` have much longer lifetimes than the `Iterators` created from them. Because of this, JavaMOP would have large numbers of monitor instances—when monitoring most programs—that could never possibly match the pattern because their bound `Iterators` had been collected. The RV system, which is presented in this paper, is a commercial grade system

developed by Runtime Verification, Inc. RV is able to collect these monitor instances, as well as many others that JavaMOP does not collect.

To collect monitor instances that JavaMOP is unable to collect, we implement, in RV, a means to prune unnecessary monitor instances based on a static analysis of the monitored property. The results of the static analysis, which we refer to as coenable sets, are used at runtime to determine when a monitor instance can no longer reach a triggering state, and can thus be garbage collected. For example, in UNSAFEITER (Figure 17.3), the coenable sets associated to event `update` consist of all those subsets of events which can potentially make `update` a relevant event for a monitor for UNSAFEITER, that is, $\{\text{next}\}$, $\{\text{next}, \text{update}\}$, and $\{\text{next}, \text{create}, \text{update}\}$. Indeed, in any matching trace containing `update`, the event `update` is followed by precisely all the events in one of these subsets. Consider now a monitor M for UNSAFEITER corresponding to a particular parameters instance, say $c \mapsto c_1$ and $i \mapsto i_1$, and suppose that an event `update` is just being dispatched to M . At this moment, M knows that it has a future only if all the events in at least one of the coenable sets of `update` are possible. In particular, if the `Iterator` i_1 has already been garbage collected, then M will never `match`, since each of the coenable sets of `update` contains a `next`, which can only be generated by i_1 . Thus, M can safely terminate itself and be garbage collected in this situation. Removing unnecessary monitors is still an expensive task, and in the interest of making it as efficient as possible, a lazy collection method is used. This technique makes RV the most efficient parametric monitoring system to date, by a large margin (see Section 17.5).

Our monitor garbage collection technique is orthogonal to other optimization techniques for parametric monitoring. More precisely, our technique is aimed at improving the base performance of parametric monitoring by means of keeping the number of monitor instances low *without* relying on (expensive) knowledge about the source program or on minimizing the distance between events and their monitors. Other optimizations can be applied on top of our garbage collection technique and thus start from this base performance and improve it. For example, staged indexing (or decentralized indexing), which has been proposed and implemented in [25, 62, 22], piggybacks indexing trees onto parameter instances. This reduces the cost of lookup due to better cache locality and fewer hash lookups. Also, significant runtime overhead reductions have been achieved using program static analysis [192, 42, 40, 93], by removing unnecessary instrumentation. RV supports both staged indexing and program static analysis via the Clara approach [42].

Nevertheless, *we deliberately disabled these orthogonal optimizations in our evaluation, to properly measure the effectiveness of the proposed garbage collection technique.* Enabling these orthogonal optimizations would only hide the inefficiency of base monitoring.

We evaluate our RV garbage collection technique and compare it to those in JavaMOP and Tracematches in Section 17.5. We picked these two systems for comparison because they are known for their efficiency (the best so far). The average overhead of RV in version 9.12 of the DaCapo [36] benchmark suite is 15%, even with no static or decentralized indexing optimizations, which is two times lower than 33% of JavaMOP and nine times lower than the 142% of Tracematches *disregarding* those cases where Tracematches *failed to terminate*. Even the largest overhead of RV in two versions of DaCapo, from UNSAFEITER-bloat, is only 251%, while in JavaMOP, 7 cases show overhead higher than 251%, and in Tracematches, 20 cases show higher overhead and 9 cases do not terminate.

Outline The rest of this paper is as follows: Section 17.2 provides a brief overview of parametric monitoring; Section 17.3 explains the theory of the coenables sets used for pruning unnecessary monitor instances, and shows some examples; Section 17.4 discusses our data structures to efficiently garbage collect monitors by using coenables sets, as well as how the coenable sets are actually used during the monitoring process; Section 17.5 presents our experimental data; and Section 17.6 provides some concluding remarks.

17.2 Parametric Properties and Monitoring

To explain the garbage collection of unnecessary monitor instances, we first introduce some background theory on parametric monitoring. For consistency, we follow the notation and terminology recently proposed by the JavaMOP authors in [65]. We begin by introducing the notions of event, trace, and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets events that are unrelated to the given parameter instance.

Definition 100 *Let \mathcal{E} be a finite set of (non-parametric) events, called **base events** or simply **events**. An \mathcal{E} -trace, or simply a (non-parametric) **trace** when \mathcal{E} is understood or not important, is any finite sequence of events in \mathcal{E} , that is, an element in \mathcal{E}^* . If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$. ϵ is the empty trace.*

For UNSAFEITER in Section 17.1, the set of events \mathcal{E} is $\{\text{create}, \text{update}, \text{next}\}$, and a possible trace is “create next update next”.

Definition 101 *An \mathcal{E} -property P , or simply a (base or non-parametric) **property**, is a function $P: \mathcal{E}^* \rightarrow \mathcal{C}$ partitioning the set of traces into (**verdict**) **categories** \mathcal{C} . In general, \mathcal{C} may be any set.*

Consider again UNSAFEITER. The **match** traces are those matching the pattern, e.g., “create next update next”. There are also traces that have not matched yet, but may still match in the future, such as “update create”, which we call $?$ (unknown) traces. Lastly, there are traces that may never match again, such as “create update next next”, which we refer to as **fail** traces. Thus we pick \mathcal{C} to be the set $\{\text{match}, \text{fail}, ?\}$, and define its property $P_{\text{UNSAFEITER}}: \mathcal{E}^* \rightarrow \mathcal{C}$ as follows: $P_{\text{UNSAFEITER}}(w) = \text{match}$ if w is in the language of the UNSAFEITER ere, $P_{\text{UNSAFEITER}}(w) = ?$ if w is a prefix of a string in the language of the ere, and $P_{\text{UNSAFEITER}}(w) = \text{fail}$ otherwise.

We next extend the above definitions to the parametric case. Let $[A \rightarrow B]$ be the set of total functions, and let $[A \dashrightarrow B]$ be the set of partial functions from A to B .

Definition 102 (Parametric events and traces). *Let X be a finite set of **parameters** and let V be a set of corresponding **parameter values**. If \mathcal{E} is a set of base events like in Definition 100, then let $\mathcal{E}\langle X \rangle$ be the set of corresponding **parametric events** $e\langle \theta \rangle$, where e is a base event in \mathcal{E} and θ is a partial function in $[X \dashrightarrow V]$. Partial functions θ in $[X \dashrightarrow V]$ are called **parameter instances**. A **parametric trace** is a trace with events in $\mathcal{E}\langle X \rangle$, that is, a word in $\mathcal{E}\langle X \rangle^*$.*

A parametric trace for UNSAFEITER could be “update $\langle c \mapsto c_1 \rangle$ update $\langle c \mapsto c_2 \rangle$ create $\langle c \mapsto c_1, i \mapsto i_1 \rangle$ next $\langle i \mapsto i_1 \rangle$ ”. To simplify writing we often assume the parameter set implicit, as in the following, which is the same trace: “update $\langle c_1 \rangle$ update $\langle c_2 \rangle$ create $\langle c_1, i_1 \rangle$ next $\langle i_1 \rangle$ ”.

Definition 103 *Let X be a finite set of parameters. If \mathcal{E} is a set of base events like in Definition 100, we define a **parametric event definition**, or **event definition** for short, as a function $\mathcal{D}: \mathcal{E} \rightarrow \mathcal{P}(X)$, where \mathcal{P} is the power set, that maps each event e to a set of parameters $\mathcal{D}(e)$ that will be instantiated by e at runtime. \mathcal{D} is extended to \mathcal{E}^* as $\mathcal{D}(\epsilon) = \emptyset$ and $\mathcal{D}(ew) = \mathcal{D}(e) \cup \mathcal{D}(w)$, and to $\mathcal{P}(\mathcal{E})$ as $\mathcal{D}(\emptyset) = \emptyset$ and $\mathcal{D}(\{e\} \cup E) = \mathcal{D}(e) \cup \mathcal{D}(E)$. Parametric event $e\langle \theta \rangle$ is **\mathcal{D} -consistent** if $\text{Dom}(\theta) = \mathcal{D}(e)$. Parametric trace τ is **\mathcal{D} -consistent** if $e\langle \theta \rangle$ is \mathcal{D} -consistent for each $e\langle \theta \rangle \in \tau$.*

The UNSAFEITER property contains the parametric event definition $\mathcal{D}(\text{create}) = \{c, i\}$, $\mathcal{D}(\text{update}) = \{c\}$, $\mathcal{D}(\text{next}) = \{i\}$. It states that, for example, parameters c and i will be instantiated at runtime when a parametric event $\text{create}\langle\theta\rangle$ is received. For a trace “create update”, $\mathcal{D}(\text{create update})$ is $\{c, i\}$.

Definition 104 $\theta, \theta' \in [A \rightarrow B]$ are **compatible** if for any $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances θ and θ' , written $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{if } \theta(x) \text{ is defined} \\ \theta'(x) & \text{if } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\theta \sqcup \theta'$ is also called the **least upper bound (lub)** of θ and θ' . θ is **less informative** than θ' , written $\theta \sqsubseteq \theta'$, if for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$. \sqcup is extended to $\mathcal{P}_f([X \rightarrow V])$ in the natural way. Here \mathcal{P}_f is the finite power set.

Definition 105 (Trace slicing) Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and θ in $[X \rightarrow V]$, let the θ -**trace slice** $\tau_\theta \in \mathcal{E}^*$ be the non-parametric trace defined as:

- $\epsilon_\theta = \epsilon$ (recall that ϵ is the empty trace)
- $(\tau e\langle\theta'\rangle)_\theta = \begin{cases} (\tau_\theta)e & \text{if } \theta' \sqsubseteq \theta \\ \tau_\theta & \text{otherwise} \end{cases}$

The trace slice τ_θ first filters out all the parametric events that are not relevant for the instance θ , i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ , it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard events from parameter instances that are not relevant to θ during the slicing, including those more informative than θ . Referring back to our parametric trace from above, the non-parametric trace slice for parameter instance $\langle c_2 \rangle$ is “update”, that for $\langle c_1 \rangle$ is “update”, the slice for $\langle c_1, i_1 \rangle$ is “update next”, and the slice for $\langle i_1 \rangle$ is “next”.

Definition 106 Let X be a finite set of parameters together with their corresponding parameter values V , like in Definition 102, and let $P: \mathcal{E}^* \rightarrow \mathcal{C}$ be a non-parametric property like in Definition 101. Then we define the

parametric property $\Lambda X . P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and verdict categories $[[X \rightarrow V] \rightarrow \mathcal{C}]$)

$$\Lambda X . P : \mathcal{E}\langle X \rangle^* \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$$

as $(\Lambda X . P)(\tau)(\theta) = P(\tau_\theta)$ for each $\tau \in \mathcal{E}\langle X \rangle^*$, $\theta \in [X \rightarrow V]$.

A parametric property is therefore similar to a normal property, but one partitioning parametric traces in $\mathcal{E}\langle X \rangle^*$ into verdict categories in $[[X \rightarrow V] \rightarrow \mathcal{C}]$, that is, original (as in the non-parametric property) verdict categories indexed by parameter instances. This allows the parametric property to associate an original category for each parameter instance from $[X \rightarrow V]$.

Next we define monitors and parametric monitors. Like for parametric properties, which are just properties over parametric traces, parametric monitors are also just monitors, but for parametric events and with instance-indexed states and verdict categories.

Definition 107 A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, \mathbf{1}, \sigma, \gamma)$, where S is the set of states, \mathcal{E} is the set of input events, \mathcal{C} is the set of verdict categories, $\mathbf{1} \in S$ is the initial state, $\sigma : S \times \mathcal{E} \rightarrow S$ is the transition function, and $\gamma : S \rightarrow \mathcal{C}$ is the verdict function. The transition function is extended to handle traces, i.e., $\sigma : S \times \mathcal{E}^* \rightarrow S$ where $\sigma(s, \epsilon) = s$ and $\sigma(s, ew) = \sigma(\sigma(s, e), w)$. $M = (S, \mathcal{E}, \mathcal{C}, \mathbf{1}, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \rightarrow \mathcal{C}$ if $\gamma(\sigma(\mathbf{1}, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Monitor M defines the property $P_M : \mathcal{E}^* \rightarrow \mathcal{C}$ with $P_M(w) = \gamma(\sigma(\mathbf{1}, w))$. Monitors M and M' are equivalent iff $P_M = P_{M'}$.

We next define parametric monitors starting with a base monitor and a set of parameters: the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 108 Given parameters X with corresponding values V and monitor $M = (S, \mathcal{E}, \mathcal{C}, \mathbf{1}, \sigma, \gamma)$, the **parametric monitor** $\Lambda X . M$ is the monitor $([[X \rightarrow V] \rightarrow S], \mathcal{E}\langle X \rangle, [[X \rightarrow V] \rightarrow \mathcal{C}], \lambda \theta. \mathbf{1}, \Lambda X . \sigma, \Lambda X . \gamma)$, with

- $\Lambda X . \sigma : [[X \rightarrow V] \rightarrow S] \times \mathcal{E}\langle X \rangle \rightarrow [[X \rightarrow V] \rightarrow S]$
- $\Lambda X . \gamma : [[X \rightarrow V] \rightarrow S] \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$

```

Algorithm MONITOR( $M = (S, \mathcal{E}, \mathcal{C}, \mathbf{i}, \sigma, \gamma)$ )
function main( $\tau$ )
1  $\Delta \leftarrow \perp$ ;  $\Delta(\perp) \leftarrow \mathbf{i}$ ;  $\Theta \leftarrow \{\perp\}$ 
2 foreach  $e\langle\theta\rangle$  in order in  $\tau$  do
3 : foreach  $\theta' \in \{\theta\} \sqcup \Theta$  do
4 :    $\Delta(\theta') \leftarrow \sigma(\Delta(\max\{\theta'' \in \Theta \mid \theta'' \sqsubseteq \theta'\}), e)$ 
5 :    $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$ 
6 : endfor
7 :  $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$ 
8 endfor

```

Figure 17.5: Monitoring Algorithm

defined as

$$\begin{aligned}
(\Lambda X . \sigma)(\delta, e\langle\theta'\rangle)(\theta) &= \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{otherwise} \end{cases} \\
(\Lambda X . \gamma)(\delta)(\theta) &= \gamma(\delta(\theta))
\end{aligned}$$

for each $\delta \in [[X \rightarrow V] \rightarrow S]$ and each $\theta, \theta' \in [X \rightarrow V]$.

Therefore, a parametric monitor $\Lambda X . M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ , takes parametric events as input, and outputs categories indexed by parameter instances (one category of M per instance). Intuitively, one can think of a parametric monitor as a collection of “*monitor instances*”. Each monitor instance, which is indexed by a parameter instance, keeps track of the state of one trace slice. The rule for $\Lambda X . \sigma$ can be read as stating that when an event with parameter instance θ' is evaluated, it updates the state for all monitor instances more informative than the instance for θ' , and the instance for θ' itself, leaving all other monitor instances untouched. The rule for $\Lambda X . \gamma$ simply states that γ is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for θ . One of the major results in [65] states that if M is a monitor for P , parametric monitor $\Lambda X . M$ is a monitor for the parametric property $\Lambda X . P$.

Figure 17.5 shows the basic abstract monitoring algorithm for parametric properties from [65]. Given parametric property $\Lambda X . P$ and M a monitor for P , MONITOR(M) yields a monitor that is equivalent to $\Lambda X . M$, that is, a monitor for $\Lambda X . P$. The functions $[[X \rightarrow V] \rightarrow S]$ and $[[X \rightarrow V] \rightarrow \mathcal{C}]$ of $\Lambda X . M$ are encoded by MONITOR(M) as tables Δ and Γ with entries indexed by parameter instances in $[X \rightarrow V]$ and with contents states in S and verdict

categories in \mathcal{C} , respectively. Such tables will have finite entries because each event e binds only a finite number of parameters defined by $\mathcal{D}(e)$.

The monitoring algorithm first clears Δ , which contains the monitor state for each parameter instance, then assigns \perp , the initial state, to $\Delta(\perp)$. Θ , which contains all known parameter instances, is initialized to contain only the empty partial function \perp . For each event $e(\theta)$ that arrives during program execution (line 2), $\text{MONITOR}(M)$ generates every compatible parameter instance by combining θ with all the previously known compatible parameter instances (line 3). It then updates the state of every one of these compatible parameter instances (θ') with the state, transitioned by event e , of the “monitor instance” corresponding to the “largest” parameter instance less than or equal to θ' (line 4). At the same time we also calculate the verdict category corresponding to that monitor instance and store it in table Γ (line 5). Rather than storing a whole slice as in Definition 105, the knowledge of the slice is encoded in the state of the monitor instance for θ' . After the algorithm completes, Γ contains the verdict category for each possible trace slice. An actual implementation is free to report a verdict category of interest (e.g., `match` or `fail`) as soon as it is discovered.

17.3 Coenable Sets

When monitoring parametric properties, it is easy to generate a large number of monitor instances. For example, as seen in Section 17.5, the program `bloat` generates 1.9 million monitor instances when monitored for the `UNSAFEITER` property. After some time, some of these monitor instances may become unnecessary, e.g., because they have no hope of reaching a verdict category in \mathcal{G} . Indeed, as seen in Section 17.5, the RV garbage collection technique flags 1.8 million of these monitor instances as unnecessary. Chen et al. [57] proposed a formalism-independent method, called “ENABLE sets”, to avoid needlessly *creating* monitors that will never trigger. Here we show how a dual method can be derived to avoid needlessly *retaining* monitors that will never trigger. Computing the coenable sets is expected to be a quick static operation in practice, because they are a function of the specification to monitor (which is expected to be small) and not of the program (which is expected to be large).

Definition 109 *Given $w \in \mathcal{E}^*$ and $e, e' \in w$, we let $e \rightsquigarrow_w e'$ denote that e' occurs after e in w . Let $\text{COENABLE}_w(e) = \{e' \mid e \rightsquigarrow_w e'\}$ be the **trace coenable set** of e . Given property $P: \mathcal{E}^* \rightarrow \mathcal{C}$ and a subset of verdict categories*

of interest (or goal) $\mathcal{G} \subseteq \mathcal{C}$, the **property coenable set** is defined as the map $\text{COENABLE}_{P,\mathcal{G}}: \mathcal{E} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{E}))$ where $\text{COENABLE}_{P,\mathcal{G}}(e) = \{\text{COENABLE}_w(e) \mid w \in \mathcal{E}^* \text{ s.t. } P(w) \in \mathcal{G}, e \in w, \text{COENABLE}_w(e) \neq \emptyset\}$ for each $e \in \mathcal{E}$.

Intuitively, if event e is encountered during monitoring, but none of the event sets of $\text{COENABLE}_{P,\mathcal{G}}(e)$ are possible in the future, it is impossible to reach any verdict category in \mathcal{G} , so a monitor for P observing e will never trigger. We drop all \emptyset s from $\text{COENABLE}_{P,\mathcal{G}}$ because they can cause monitor instances to be retained that are unnecessary. An \emptyset in $\text{COENABLE}_{P,\mathcal{G}}(e)$ means that the trace suffix consisting of only the event e can lead to a category in \mathcal{G} for some trace prefix. However, our interest is in the ability to reach \mathcal{G} again in the future. If there is a trace suffix that can lead to a state in \mathcal{G} from e , then its events will be added to $\text{COENABLE}_{P,\mathcal{G}}(e)$. If there is no trace suffix that can lead back to a state in \mathcal{G} , there is no reason to maintain the monitor instance after it has executed the proper handler due to the occurrence of e . **FSM Example** We define finite state machines in the spirit of Definition 107. A finite state machine is a tuple $(S, \mathcal{E}, \mathcal{C}, \beta, \sigma, \gamma)$ where \mathcal{E} is a finite alphabet, S is a finite set of states, $\beta \in S$ is the initial state, $\sigma: S \times \mathcal{E} \rightarrow S$ a partial transition function, \mathcal{C} a set of verdict categories, and $\gamma: S \rightarrow \mathcal{C}$ the verdict function. The property monitored by an FSM classifies a trace w into $\gamma(\sigma(\beta, w))$, where σ is extended to strings in the natural way, and fail if $\sigma(\beta, w)$ is undefined.

We can find $\text{COENABLE}_{P,\mathcal{G}}$, for the property monitored by an FSM, by the least fixed point of the following equations. Recall that $\mathcal{G} \subseteq \mathcal{C}$ is the set of verdict categories of interest:

$$\begin{aligned} \text{SEEABLE}(s) &= \bigcup_{\sigma(s,e)=s'} \{\{e\} \cup T \mid T \in \text{SEEABLE}(s')\} \\ \text{COENABLE}_{P,\mathcal{G}}(e) &= \bigcup_{\sigma(s,e)=s'} \text{SEEABLE}(s') \end{aligned}$$

We can use the equations above to generate coenable sets for our example from Figure 17.3, one need simply generate a finite state machine from the property's ERE. For $P = \text{UNSAFEITER}$ and $\mathcal{G} = \{\text{match}\}$, the $\text{COENABLE}_{P,\mathcal{G}}$ sets are:

$$\begin{aligned} \text{COENABLE}_{P,\mathcal{G}}(\text{create}) &= \{\{\text{next}, \text{update}\}\} \\ \text{COENABLE}_{P,\mathcal{G}}(\text{update}) &= \left\{ \{\text{next}\}, \{\text{next}, \text{update}\}, \right. \\ &\quad \left. \{\text{next}, \text{create}, \text{update}\} \right\} \\ \text{COENABLE}_{P,\mathcal{G}}(\text{next}) &= \{\{\text{next}, \text{update}\}\} \end{aligned}$$

Note that if we did not remove \emptyset s, $\text{COENABLE}_{P,\mathcal{G}}(\text{next})$ would contain \emptyset . Each inner set can be thought of as a conjunction of events that must occur at least once for a verdict category in \mathcal{G} to still be reachable, while the outer sets are a disjunction (see Section 17.4.2). For example, if the event seen by monitor instance M is `update` and `next` can still be seen at some future point, then M is still necessary. Likewise, if the event seen by M is `next`, then both `next` and `update` must be possible for M to ever `match`. In particular, if the corresponding `Collection` object instance is already dead then we know that the event `update` will never be possible, so we can safely garbage collect M . Definition 110 formalizes this notion.

CFG Example A CFG is a tuple $(N, \mathcal{E}, \mathbf{S}, \Pi)$ where N is a finite set of nonterminals, \mathcal{E} is a finite set of terminals, $\mathbf{S} \in N$ is the initial nonterminal, and Π is a set of productions of the form $\mathbf{A} \rightarrow \beta$ where $\mathbf{A} \in N$ and $\beta \in (N \cup \mathcal{E})^*$. The monitor for a CFG classifies traces that are in the language of the grammar into the verdict category `match`.

For a CFG, to compute $\text{COENABLE}_{P,\{\text{match}\}}$ we find the least fixed point of the following equations:

$$\begin{aligned} G(\epsilon) &= \{\emptyset\} & G(e) &= \{\{e\}\} & G(\mathbf{A}) &= \bigcup_{\mathbf{A} \rightarrow \beta} G(\beta) \\ G(\beta_1\beta_2) &= \{T_1 \cup T_2 \mid T_1 \in G(\beta_1), T_2 \in G(\beta_2)\} \\ C(x) &= \left\{ T_1 \cup T_2 \mid \begin{array}{l} \mathbf{A} \rightarrow \beta_1 x \beta_2, \\ T_1 \in C(\mathbf{A}), T_2 \in G(\beta_2) \end{array} \right\} \\ \text{COENABLE}_{P,\{\text{match}\}}(e) &= C(e) \end{aligned}$$

Informally, $G(\mathbf{A})$ is the set of events generated by the CFG, if the symbol \mathbf{A} were used as the initial nonterminal of the CFG. The equation $G(\beta_1\beta_2) = \{T_1 \cup T_2 \mid T_1 \in G(\beta_1), T_2 \in G(\beta_2)\}$ generalizes this notion to entire traces of symbols (where symbols are either events or non-terminals). C is the coenable sets function generalized to traces that include both non-terminals and events. For a production, $\mathbf{A} \rightarrow \beta_1 \mathbf{B} \beta_2$, $C(\mathbf{B})$ needs to cope with the fact that \mathbf{A} has its own coenable sets. Thus its definition unions possible coenable sets of \mathbf{A} with the sets of symbols that are generated by β_2 . The rest of RV only needs to know coenable sets for events so coenables is just the restriction of C to events.

Definition 110 Given property $P: \mathcal{E}^* \rightarrow \mathcal{C}$, goal $\mathcal{G} \subseteq \mathcal{C}$, set of parameters X and event definition $\mathcal{D}: \mathcal{E} \rightarrow \mathcal{P}(X)$ (see Definition 103), the **property parameter coenable set** is defined as the map $\text{COENABLE}_{P,\mathcal{G}}^X: \mathcal{E} \rightarrow \mathcal{P}(\mathcal{P}(X))$ where $\text{COENABLE}_{P,\mathcal{G}}^X(e) = \{\mathcal{D}(E) \mid E \in \text{COENABLE}_{P,\mathcal{G}}(e)\}$ for each $e \in \mathcal{E}$.

The $\text{COENABLE}_{P,\mathcal{G}}^X$ sets tell us which parameter objects must be alive for a verdict category in \mathcal{G} to be reachable. For $P = \text{UNSAFEITER}$, $\mathcal{G} = \{\text{match}\}$, and $X = \{c, i\}$, the $\text{COENABLE}_{P,\mathcal{G}}^X$ sets are:

$$\begin{aligned}\text{COENABLE}_{P,\mathcal{G}}^X(\text{create}) &= \{\{c, i\}\} \\ \text{COENABLE}_{P,\mathcal{G}}^X(\text{update}) &= \{\{i\}, \{c, i\}\} \\ \text{COENABLE}_{P,\mathcal{G}}^X(\text{next}) &= \{\{c, i\}\}\end{aligned}$$

Now with the $\text{COENABLE}_{P,\mathcal{G}}^X$ sets we can explicitly decide when a monitor instance may be collected. For example, in UNSAFEITER we know that if, at *any time*, the `Iterator` bound to i is garbage collected, then a `match` can never occur because i occurs in every one of the inner sets. This makes sense because the event that causes a match in the UNSAFEITER pattern is use of the `Iterator`. As mentioned in Section 17.1, this situation could produce a very large memory leak in `JavaMOP` [62] where long living `Collections` would cause monitor instances for dead `Iterators` to be retained because it could not remove a monitor instance unless all bound parameter objects were collected. We prove this concept by showing that certain parameters specified by $\text{COENABLE}_{P,\mathcal{G}}^X(e)$ for a trace wew' must be able to occur in w' for a verdict category to be reached.

Theorem 27 *Consider the same assumptions as in Definition 110, and a trace slice $wew' \in \mathcal{E}^*$. If for each $Y \in \text{COENABLE}_{P,\mathcal{G}}^X(e)$ there exists some $y \in Y$ such that $y \notin \mathcal{D}(w')$ then $P(wew') \notin \mathcal{G}$.*

Proof: Suppose, for the sake of contradiction, that $P(wew') \in \mathcal{G}$ and that each $Y \in \text{COENABLE}_{P,\mathcal{G}}^X(e)$ contains a y such that $y \notin \mathcal{D}(w')$. By Definition 109, because $P(wew') \in \mathcal{G}$ there must be some $E \in \text{COENABLE}_{P,\mathcal{G}}(e)$ that contains exactly those events in w' . Then, by Definition 110, there must be $Y \in \text{COENABLE}_{P,\mathcal{G}}^X(e)$ containing exactly the parameters in $\mathcal{D}(w')$. Contradiction. □ **Discussion**

The $\text{COENABLE}_{P,\mathcal{G}}^X$ sets are a conservative approximation of the situations in which a monitor instance may be collected. From Definition 105 we know that an event e where $x \in \mathcal{D}(e)$ can only occur in a trace-slice π_θ if $\theta(x)$ is still alive in the system. If $\theta(x)$ has been garbage collected, there is no way for any e with $x \in \mathcal{D}(e)$ to occur in trace slice for θ . This is precisely how monitoring arrives in the situation presented in Theorem 27, where all possible suffixes w' of the trace slice wew' do not contain at least one parameter in each set of the $\text{COENABLE}_{P,\mathcal{G}}^X(e)$, and it becomes impossible to

reach a verdict category in \mathcal{G} . Clearly, if it is impossible for the θ trace slice to ever reach a verdict category in \mathcal{G} , there is no reason to keep the monitor instance for θ .

The Tracematches system uses a more precise formulation, which is similar, but based on the *state* of the monitor. Intuitively, the Tracematches garbage collection technique can be thought of as coenables sets indexed by state rather than events, but the formulation as presented in [25] is considerably different. While theirs is more precise, our empirical results, presented in Section 17.5, show that the coenable set technique is able to reduce memory usage in the RV system to comparable levels with Tracematches, while the RV system has considerably lower runtime overhead. More importantly, the Tracematches garbage collection technique is limited to finite logics, such as the regular expressions of Tracematches. However, our coenable approach is extensible to any underlying monitor implementation. We have a coenables sets generation algorithm for the context-free grammar plugin. A static state-based technique, such as the one used by Tracematches, could not be used for context-free properties because the state space is unbounded.

The coenables technique reclaims much more memory than JavaMOP’s garbage collection, which, as already explained, has to wait for all bound parameter objects to be collected (see Section 17.5).

17.4 Implementation

The data structures used by previous runtime monitoring systems [9, 83, 62] are not sufficient for efficient garbage collection of monitor instances. The challenge is how to efficiently garbage collect unnecessary monitor instances that are contained in the data structures. Using the standard data structures of previous systems, the overhead of instance removal easily overwhelms the benefit of having fewer instances. Our specialized data structures, introduced here, track the garbage collection of parameter objects and remove unnecessary monitor instances when discovered using coenable sets (Section 17.3). In this section, we present the modified indexing trees used by RV as well as the mechanism by which unnecessary monitors are garbage collected.

17.4.1 Indexing Trees

The RV system builds upon the Indexing Tree technique of the JavaMOP system presented in [62]. The indexing trees are an efficient means to rep-

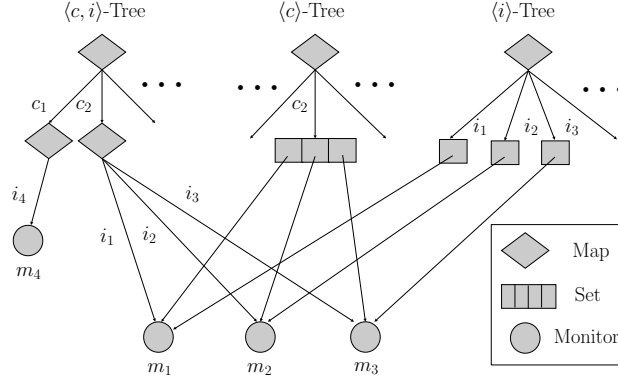


Figure 17.6: Indexing trees for UNSAFEITER

represent the tables Δ and Γ from the monitoring algorithm in Figure 17.5. Locating the correct monitor instances to update for each received event is one of the most important and expensive tasks of a runtime monitoring system that supports formalism-independent parametric properties like JavaMOP and RV. Whenever a parametric event $e\langle\theta\rangle$ is processed, all monitor instances corresponding to parameter instances more informative than θ must be updated. Thus we need a mapping from parameter instances to sets of monitor instances more informative than that instance. Each value in the map is either the next level of the tree or, at the leaves, the appropriate set of monitor instances. Once we have this set we update all the contained monitor instances.

For example, processing $\text{update}\langle c_2 \rangle$ for UNSAFEITER (Figure 17.3) requires that we update the monitor instances that are more informative than $\langle c_2 \rangle$. Thus we lookup $\langle c_2 \rangle$ in the $\langle c \rangle$ -tree of Figure 17.6 to find the set of monitor instances more informative than $\langle c_2 \rangle$ and update each instance. Multiple indexing trees can exist since each event may contain a different subset of parameters; each subset of possible parameters receives its own tree. As an example, for UNSAFEITER we have a $\langle c \rangle$ -tree, an $\langle i \rangle$ -tree, and a $\langle c, i \rangle$ -tree.

17.4.2 Collecting Unnecessary Monitors

There are two performance benefits to garbage collecting unnecessary monitors: reduced memory usage, and reduction in the time needed to update monitor instances because many of the monitor instances that would be updated are no longer necessary. As an example of the latter, consider

UNSAFEITER again. If we have a monitor instance for $\langle c_1, i_1 \rangle$ and i_1 is garbage collected, even though it is impossible to match the UNSAFEITER pattern, all future `update` events are sent to $\langle c_1, i_1 \rangle$. Unfortunately, collecting monitor instances introduces overhead; we must keep this overhead low so that it does not outweigh the benefits of garbage collection.

Eager garbage collection of unnecessary monitors introduces a very large amount of runtime overhead, which almost always overwhelms any benefits. This is because eager collection requires propagating the information regarding liveness of parameter objects to monitor instances far too frequently. Additionally, eager collection can result in removing instances from some data structures that will never be used again.

Therefore, we use a lazy garbage collection scheme. We iterate monitor instances and propagate the information of garbage collections of parameter objects *lazily*, and we remove unnecessary monitors *lazily*. When an indexing tree containing a garbage collected parameter object is accessed and the tree detects this, it informs all the relevant monitor instances contained within itself. Then, the monitor instance decides if it can still possibly reach a verdict category in \mathcal{G} in the absence of the parameter object that has been garbage collected. Later when more space is needed in the data structure or when monitor instances are updated, we remove monitor instances from the accessed data structure but not from other data structures. A monitor instance is garbage collected when it is removed from all data structures. This is similar to mark-and-sweep garbage collection. If a data structure itself is garbage collected, any contained monitor instances never need to be garbage collected separately. The next sections explain this process in detail.

Parameter Object Garbage Collection Notification

Propagation of parameter object garbage collection information starts from the mappings in the indexing tree. The mappings used in the RV are implemented as a class called `RVMap`. `RVMap` uses `WeakReferences` for its keys. A `WeakReference` in Java does not stop the garbage collector from collecting its referent; when the referent is garbage collected, the `WeakReference` points to `null`. Whenever an operation (`put` or `get`) is performed on an `RVMap`—or the hash table underlying the map needs to be expanded to store more entries—it looks through a subset of its entries for keys with `null` referents. When there is a key with a `null` referent due to a garbage collection, `RVMap` notifies all of the monitor instances below itself in the indexing tree. For example, Figure 17.7 (A) shows a possible scenario where $\langle c_2 \rangle$ is garbage

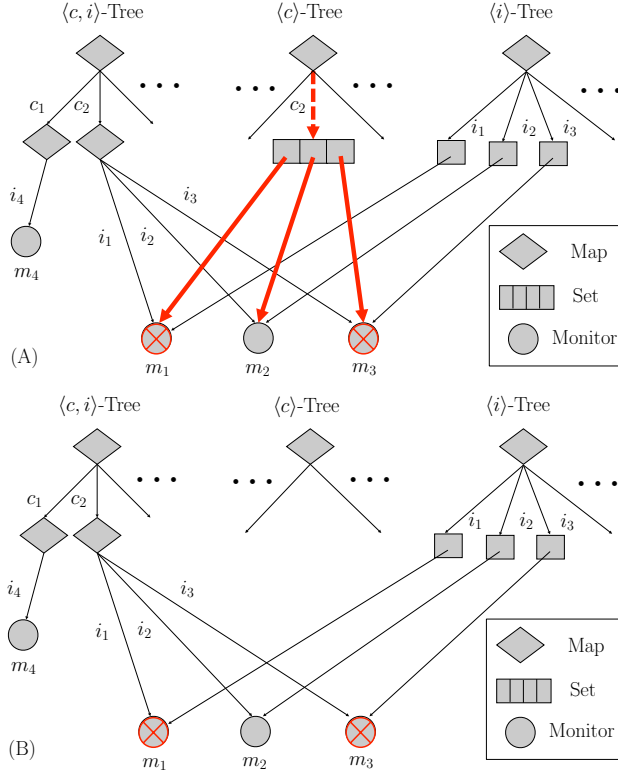


Figure 17.7: (A) Notifying monitors for garbage collected $\langle c_2 \rangle$ in the $\langle c \rangle$ -tree. (B) Cleaning up the broken mapping in the $\langle c \rangle$ -tree

collected and the $\langle c \rangle$ -tree is accessed. The $\langle c \rangle$ -tree notifies all of the monitor instances below $\langle c_2 \rangle$.

Determining When Monitor Instances are Unnecessary

When a monitor is notified of a newly garbage collected parameter object, it decides whether it can still reach a verdict category of interest in the absence of garbage collected parameter objects by using the coenable sets introduced in Section 17.3. Each monitor instance stores the last event it receives, e , so that it may check $\text{COENABLE}_{P,G}^X(e)$, when this notification takes place. The monitor instance need simply check if all the parameter objects of any set in $\text{COENABLE}_{P,G}^X(e)$ are alive. RV statically translates $\text{COENABLE}_{P,G}^X(e)$ to a

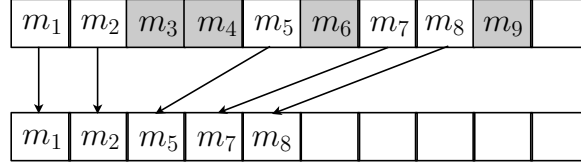


Figure 17.8: A compaction in *RVSet* when some monitor instances are collectable

minimized boolean formula to make this check as efficient as possible:

$$\text{ALIVENESS}(e) = \bigvee_{S \in \text{COENABLE}_{P,G}^X(e)} \left(\bigwedge_{x \in S} \text{live}_x \right)$$

where live_x is a boolean that is true only if the parameter object of parameter x has not been garbage collected. Then, $\text{ALIVENESS}(e)$ is true only if the monitor is necessary. Maintaining live_x variables in a given monitor instance for each parameter and checking the generated boolean expression at runtime is sufficient for determining when said instance becomes unnecessary.

Continuing our example from the last section, the monitor instances notified of garbage collected parameters in Figure 17.7 (A) check their ALIVENESS to determine if they are unnecessary. Here, m_1 and m_3 are unnecessary and therefore marked. Note that the set under $\langle c_2 \rangle$ is not altered because other *RVMaps* in the index tree still point to it. In Figure 17.7 (B), the *RVMap* removed the broken map entry index by c_2 . m_1 and m_3 will be removed at some future time when the $\langle c, i \rangle$ -tree or $\langle i \rangle$ -tree are accessed or expanded, as we explain in the next section.

Removing Unnecessary Monitor Instances

Monitor instances are removed lazily because in many cases the maps and sets containing monitor instances flagged for removal may be garbage collected themselves. Eager removal would result in unnecessary work in such cases. For example, in Figure 17.7 (B), if the $\langle c_2 \rangle$ -subtree in the $\langle c, i \rangle$ -tree is going to be garbage collected, there is no reason to remove flagged monitor instances from it.

Unnecessary monitor instances are only removed when an indexing tree is accessed. Whenever an *RVMap* looks for keys with `null` referents it also

		HasNext			UnsafeIter			UnsafeMapIter			UnsafeSyncColl			UnsafeSyncM	
(A)	ORIG (sec)	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP
bloat	3.6	2119	448	116	19194	569	251	∞	1203	178	1359	746	212	1942	716
jython	8.9	13	0	0	11	0	1	150	18	3	11	1	1	10	0
avrrora	13.6	45	54	55	637	311	118	∞	113	42	75	144	80	54	74
batik	3.5	3	2	3	355	9	8	∞	8	5	208	9	9	5	3
eclipse	79.0	-2	4	-1	0	-1	-1	5	-3	0	-4	2	1	∞	-1
fop	2.0	200	49	48	350	21	13	∞	58	14	∞	78	25	∞	71
h2	18.7	89	17	13	128	9	4	1350	21	6	868	21	4	83	20
luindex	2.9	0	0	1	0	0	1	1	4	1	1	1	1	2	0
lusearch	25.3	-1	1	0	1	2	2	2	2	0	4	0	1	3	1
pmd	8.3	176	84	59	1423	162	123	∞	571	188	1818	192	76	∞	144
sunflow	32.7	47	5	3	7	2	0	9	4	1	13	6	5	17	6
tomcat	13.8	8	1	1	37	1	1	3	1	1	2	0	1	2	1
tradebeans	45.5	0	-1	1	1	1	2	5	3	-1	-1	1	2	3	1
tradesoap	94.4	1	3	0	2	1	1	2	0	1	0	0	1	2	2
xalan	20.3	4	2	2	27	7	2	10	5	2	3	2	3	4	4
(B)	ORIG (MB)	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP	RV	TM	MOP
bloat	4.9	56.8	19.3	13.2	7.7	146.8	79.0	∞	173.4	56.1	6.8	127.9	48.3	6.9	55.4
jython	5.3	5.7	4.6	4.8	4.9	4.6	4.8	6.0	19.5	4.7	5.3	4.5	4.4	5.9	4.8
avrrora	4.7	4.6	12.4	9.1	4.4	136.2	15.8	∞	14.7	8.5	4.3	28.0	12.6	4.4	13.0
batik	79.1	79.2	78.7	79.3	75.2	93.6	86.6	∞	91.2	79.6	78.2	93.2	85.1	79.9	86.9
eclipse	95.9	100.8	107.6	97.1	98.3	100.0	110.3	106.9	93.8	101.1	100.4	109.2	90.1	∞	98.6
fop	20.7	97.4	47.1	52.5	24.3	24.2	29.4	∞	69.2	28.1	∞	54.8	24.8	∞	55.9
h2	265.0	267.8	598.5	565.2	267.2	266.2	262.4	312.4	688.3	268.2	271.4	690.3	265.5	271.0	718.3
luindex	6.8	5.6	5.5	5.6	6.3	6.9	6.8	7.4	8.2	6.9	7.4	7.4	7.5	7.1	7.4
lusearch	4.6	4.7	4.4	4.8	4.6	4.8	4.2	4.0	4.3	4.8	4.5	4.5	4.6	4.6	4.8
pmd	18.0	56.9	59.8	48.5	17.2	146.3	86.4	∞	212.7	93.6	20.3	238.4	84.6	∞	117.1
sunflow	4.4	4.5	4.8	4.9	4.8	4.3	4.7	4.7	4.4	4.4	5.1	4.3	4.9	4.5	4.7
tomcat	11.6	11.4	12.3	11.4	12.5	11.0	11.5	11.9	11.4	11.0	11.3	11.3	11.3	11.4	11.4
tradebeans	63.2	62.9	62.7	62.1	63.7	63.9	64.1	63.3	62.5	62.7	63.2	62.8	62.0	64.0	62.8
tradesoap	64.1	61.8	62.3	63.3	63.4	63.1	64.4	64.1	63.5	62.0	60.7	65.0	65.9	65.5	64.5
xalan	4.9	4.9	5.0	5.1	4.9	4.9	4.9	4.9	4.5	4.9	5.0	4.8	5.0	5.1	4.9

Figure 17.9: Comparison of Tracematches (TM), JavaMOP (MOP), and RV:
 (A) average *percent* runtime overhead; (B) total peak memory usage in MB.
 (convergence within 3%, ∞: not terminated after 1 hour)

checks the values of mappings which do not have null referents. The value can be either a monitor instance, a set, or a lower level map. If the value is a flagged monitor instance or an empty data structure, it removes the mapping. If it is a set, it must be checked for internal monitor instances that have been flagged for removal. When a set is checked for unnecessary monitor instances, all of the instances are collected, and the remaining necessary

monitor instances are compacted in one pass, as can be seen in Figure 17.8.

17.5 Evaluation of the RV System

We evaluate our formalism-independent garbage collection for parametric monitoring implemented into RV. Also, we compare the performance of RV to JavaMOP and Tracematches, two of the most optimized monitoring systems in runtime and memory, respectively.

17.5.1 Experimental Settings

For our experiments, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [36], the most up-to-date version. We also present the result from the previous version, 2006-10 MR2 of DaCapo (DaCapo 2006-10), but only for the bloat and jython benchmarks. DaCapo 9.12 does not provide the bloat benchmark from the DaCapo 2006-10, which we favor because it generates large overheads when monitoring `Iterator`-based properties. The bloat benchmark with the UNSAFEITER specification causes 19194% runtime overhead (i.e., 192 times slower) and uses 7.7MB of heap memory in Tracematches, and causes 569% runtime overhead and uses 147MB in JavaMOP, while the original program uses only 4.9MB. Also, although the DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error. Thus, we present the result of jython from the DaCapo 2006-10. The default data input for DaCapo was used and the `-converge` option to obtain the numbers after convergence within $\pm 3\%$. We also looked into other benchmarks including Java Grande [251] and SPECjvm 2008 [3], and saw little to no overhead even with our `Iterator`-based properties. Instrumentation introduces a different garbage collection behavior in the monitored program, sometimes causing the program to slightly outperform the original program; this accounts for the negative overheads seen in both runtime and memory.

We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ compiler (ajc) version 1.6.4 is used for weaving the aspects generated by JavaMOP and RV into the target benchmarks. Another AspectJ compiler, abc [23] 1.3.0, is used for weaving Tracematches properties because Tracematches is part of abc and does not work with ajc. For JavaMOP, we used the most recent release version, 2.1.2, from the JavaMOP website [1]. For Tracematches,

	HASNEXT				UNSAFEITER				UNSAFEMAPITER				UNSAFESYNCCOLL			
	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM	E	M	FM	CM
bloat	156M	1.9M	1.9M	1.8M	81M	1.9M	1.8M	1.6M	73M	3.6M	44K	3.5M	143M	4.1M	0	3.7M
jython	106	50	47	26	179K	50	38	38	179K	101K	94	101K	156	100	0	83
avro	1.5M	909K	850K	765K	1.4M	909K	860K	808K	1.3M	1.2M	18	1.2M	2.4M	1.8M	0	1.7M
batik	49K	24K	21K	21K	125K	24K	21K	10K	55K	33K	140	27K	73K	50K	0	34K
eclipse	226K	7.6K	5.3K	2.9K	119K	6.6K	5.1K	2.6K	113K	22K	2.2K	7.8K	233K	15K	0	7.5K
fop	1.0M	184K	74K	151K	709K	7.7K	7.2K	1.8K	499K	177K	67	160K	1.2M	239K	0	217K
h2	27M	6.5M	6.0M	5.6M	12M	3.7K	3.3K	1.3K	12M	6.6M	9	6.5M	27M	6.5M	0	6.5M
luindex	371	66	40	2	4.4K	65	39	0	378	183	2	59	436	132	0	30
lusearch	1.4K	131	196	114	748K	130	210	18	20K	944	338	1.4K	1.7K	262	0	402
pmd	8.3M	789K	694K	571K	6.4M	551K	473K	382K	4.3M	1.3M	110K	1.1M	8.8M	1.5M	0	1.3M
sunflow	2.7M	101K	101K	100K	1.3M	2	0	0	1.3M	83K	0	83K	2.7M	101K	0	101K
tomcat	25	6	0	0	132	4	0	0	68	26	0	0	29	10	0	0
tradebeans	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0
tradesoap	11	3	0	0	31	2	0	0	29	13	0	0	13	5	0	0
xalan	11	3	0	0	8.9K	2	0	0	119K	20K	0	20K	13	5	0	0

Figure 17.10: Monitoring statistics: number of events (E), number of created monitors (M), number of flagged monitors (FM), number of collected monitors (CM).

we used the most recent release version, 1.3.0, from [37], which is included in the `abc` compiler as an extension. To figure out the reason that some examples do not terminate when using `Tracematches`, we also used the `abc` compiler for weaving aspects generated from RV properties. Note that RV is AspectJ compiler independent. RV shows similar overheads and terminates on all examples when using the `abc` compiler for weaving as when `ajc` is used. Because the overheads are similar, we do not present the results of using `abc` to weave RV generated aspects in this paper. However, using `abc` to weave RV properties confirms that the high overhead and non-termination come from `Tracematches` itself, not from the `abc` compiler.

The following properties are used in our experiments. They were borrowed from [40, 41, 196, 57].

- **HASNEXT**: Do not use the next element in an `Iterator` without checking for the existence of it (see Figure 17.2);
- **UNSAFEITER**: Do not update a `Collection` when using the `Iterator` interface to iterate its elements (see Figure 17.3);

- UNSAFEMAPITER: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- UNSAFESYNCCOLL: If a Collection is synchronized, then its iterator also should be accessed synchronously;
- UNSAFESYNCMAP: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner.

All of them are tested on Tracematches, JavaMOP, and RV for comparison. We also monitored all five properties at the same time in RV, which was not possible in other monitoring systems for performance reasons or structural limitations. We have tested several non-Iterator based properties: HASHSET, SAFEENUM, SAFEFILE, and SAFEFILEWRITER [40, 41, 196, 57]. None of these properties produce overheads above 5% in any of the DaCapo benchmarks, thus their results are not presented in this paper in detail. Complete results for non-Iterator based properties, as well as performance improvements for RV when combined with Clara can be found in [161].

17.5.2 Results and Discussions

Figures 17.9 and 17.10 summarize the results of the evaluation. Note that the structure of the DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for DaCapo 2006-10. Therefore, `fop` and `pmd` show higher overheads than the benchmarks using DaCapo 2006-10 from [57]. While other benchmarks show overheads less than 80% in JavaMOP, `bloat`, `avrora`, and `pmd` show prohibitive overhead in both runtime and memory performance. This is because they generate many iterators and all properties in this evaluation are intended to monitor iterators. For example, `bloat` creates 1,625,770 collections and 941,466 iterators in total while 19,605 iterators coexist at the same time at peak, in an execution. `avrora` and `pmd` also create many collections and iterators. Also, they call `hasNext()` 78,451,585 times, 1,158,152 times and 4,670,555 times and `next()` 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, we mainly discuss those three examples in this section, although RV shows improvements for other examples as well.

Figure 17.9 (A) shows the percent runtime overhead of Tracematches, JavaMOP, and RV. Overall, RV averages two times less runtime overhead than JavaMOP and orders of magnitude less runtime overhead than Tracematches

(recall that these are the most optimized runtime verification systems). With *bloat*, RV shows less than 260% runtime overhead for each property, while JavaMOP always shows over 440% runtime overhead and Tracematches always shows over 1350% for completed runs and *crashed* for UNSAFEMAPITER. With *avrora*, on average, RV shows 62% runtime overhead, while JavaMOP shows 139% runtime overhead and Tracematches shows 203% and hangs for UNSAFEMAPITER. With *pmd*, on average, RV shows 94% runtime overhead, while JavaMOP shows 231% runtime overhead and Tracematches shows 1139% and hangs for UNSAFEMAPITER and UNSAFESYNCMAP.

Also, RV was tested with all five properties together and showed 982%, 275%, and 620% overhead, respectively, which are still faster or comparable to monitoring one of many properties alone in JavaMOP or Tracematches. The overhead for monitoring all the properties simultaneously can be slightly larger than the sum of their individual overheads since the additional memory pressure makes the JVM's garbage collection behave differently.

Figure 17.9 (B) shows the peak memory usage of the three systems. RV has lower peak memory usage than JavaMOP in most cases. The cases where RV does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of RV is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [25], which cannot be implemented in a formalism-independent system like RV. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring *bloat* and *pmd*. There is a trade-off between memory usage and runtime overhead. If RV more actively removes terminated monitors, memory usage will be lower, at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient parametric monitoring system with reasonable memory performance.

Figure 17.10 shows the number of triggered events, of created monitors, of monitors flagged as unnecessary by the coenable set technique, and of monitors collected by the JVM. Among the DaCapo examples, *bloat*, *avrora*, *h2*, *pmd* and *sunflow* generated a very large number of events (more than a million) in all properties, resulting in millions of monitors created in most cases. *h2* does not exhibit large overhead because monitor instances in *h2* have shorter lifetimes, therefore the created monitor instances are not used heavily like in *bloat*. *sunflow* has millions of events but does not create as many monitor instances as as other benchmarks. When monitoring the HASNEXT and UNSAFEITER properties, the coenable sets technique effectively flagged

monitors as unnecessary and most were collected by the JVM.

17.6 Conclusion

We presented an effective novel garbage collection technique for monitoring parametric properties. Previous techniques were either completely agnostic to the property to monitor, thus incurring prohibitive runtime overheads due to memory leaks, or were intrinsically dependent on particular specification formalisms, thus being hard or impossible to use in other contexts. Our technique is the first which is both formalism-generic and efficient. As extensive evaluation shows, it is in fact significantly more efficient than the existing techniques, both formalism-generic and formalism-specific.

Our results have at least two implications. On the one hand, they show that runtime monitoring of complex specifications can be used not only for testing, but also as an integral part of the deployed system in many cases. Indeed, in most practical cases the runtime overhead is negligible, so a well-designed recovery schema implemented by means of specification handlers can ensure highly dependable systems by simply not letting them go wrong at runtime. Note that the combinations program/property selected for evaluation in this paper were specifically chosen to be bad. On the other hand, our results set a solid ground for further optimizations. For example, static analyses of the program to monitor, like those in [192, 42, 40, 93], can be used to remove unnecessary instrumentation and thus not even generate many of the monitors. Similarly, staged/decentralized indexing techniques, like those in [25, 62, 22], can reduce the distance between events and their monitors and thus reduce the overhead taken to dispatch events to monitors.

Chapter 18

Predictive Runtime Analysis

Chapter 19

Maximal Causal Models for Sequentially Consistent Systems

material from [244], including the unpublished (or published as TR) appendix (proofs and model-checking algorithm) in the original submission [79]

make sure the following will also be included: ICSE'15 [153]; PLDI'14 [154]; ICSE'08 [67]; TR'07 [66]; TR'06 [61]; TR'05 [60];

Abstract: This paper shows that it is possible to build a *maximal and sound* causal model for concurrent computations from a given execution trace. It is sound, in the sense that any program which can generate a trace can also generate all traces in its causal model. It is maximal (among sound models), in the sense that by extending the causal model of an observed trace with a new trace, the model becomes unsound: there exists a program generating the original trace which cannot generate the newly introduced trace. Thus, the maximal sound model has the property that it comprises *all* traces which *all* programs that can generate the original trace can also generate. The existence of such a model is of great theoretical value as it can be used to prove the soundness of non-maximal, and thus smaller, causal models.

19.1 Introduction

Traces of events describing concurrent computations have been employed in a plethora of methods for testing and analyzing concurrent systems. The common pattern for all these methods (e.g., [236, 27, 207, 243, 232, 102, 63, 141, 266, 246, 231, 263, 267]) is: (1) the program is instrumented to trace the execution of programs; then (2) *one* execution trace is recorded; then (3) an abstraction of that trace, i.e., a *model*, is derived; and finally, (4) the obtained model is used to “predict” (problematic) event patterns occurring in other possible executions abstracted by it.

Consider, for example, the conventional happens-before causality: if two conflicting accesses to an object are not causally ordered, then a data-race is reported [236]. But is this the best one can do? Of course, not. A series of papers propose more relaxed happens-before causal models where one can also permute blocks protected by the same lock, provided that they access disjoint variables [207], thus discovering new concurrency bugs not observable with plain happens-before. But is this the best one can do? Of course, not. Other papers propose models where one can also permute semantic blocks provided that each read access continues to correspond to the same write [243, 267, 246]. Others go even further. Section 19.5 discusses a series of existing causal models; we only study *sound* models here, i.e., ones which only report real problems in the analyzed systems, allowing developers to focus on fixing those real problems and not on additionally sorting them out from false positives. We would naturally like to know whether there is an end to the question “Is this the best we can do?”, that is, whether there is any causal model that can be associated to a given execution trace which comprises the maximum number of causally equivalent traces.

Although most runtime analysis techniques are built upon some underlying sound causal model, possibly relaxed for efficiency reasons, each effort seems to focus more on how to capture it efficiently rather than proving its soundness (often implicitly assumed) or studying its relationship to existing models (other than empirically comparing the number of found bugs). Moreover, since such approaches attempt to extract information from *one* observed trace and to find property violations, they actually deal with *causal properties* (e.g., causal data-race, causal atomicity), which are instances of desired system-wise properties that can be detected using only the causal information gathered from the observed trace. Since what can be inferred from a trace intrinsically depends on the chosen causal model, definitions of causal properties differ from technique to technique, with the undesirable

<i>Thread 1</i>	<i>Thread 2</i>	<i>Execution</i>	<i>Thread 1</i>	<i>Thread 2</i>	<i>Execution</i>
<code>sync(l) {</code>		1:	<code>sync(l) {</code>		1:
<code>y = 1;</code>		<div style="border: 1px solid black; padding: 2px;"><code>y ← 1</code></div>	<code>x = 1;</code>		<div style="border: 1px solid black; padding: 2px;"><code>x ← 1</code></div>
<code>x = 1;</code>		<div style="border: 1px solid black; padding: 2px;"><code>x ← 1</code></div>	<code>}</code>		<code>y ← 1</code>
<code>if (x == 2)</code>		<div style="border: 1px solid black; padding: 2px;"><code>x → 1</code></div>	<code>y = 1;</code>		
<code>z = 1;</code>			<code>sync(l) {</code>		<div style="border: 1px solid black; padding: 2px;"><code>x ← 1</code></div>
<code>}</code>			<code>x = 1;</code>		<code>}</code>
	<code>sync(l){</code>	2:		<code>sync(l) {</code>	2:
	<code>x = 2;</code>	<div style="border: 1px solid black; padding: 2px;"><code>x ← 2</code></div>		<code>if (x > 0)</code>	<div style="border: 1px solid black; padding: 2px;"><code>x → 1</code></div>
	<code>}</code>			<code>y = 2;</code>	<div style="border: 1px solid black; padding: 2px;"><code>y ← 2</code></div>
	<code>y = 2;</code>	<code>y ← 2</code>		<code>}</code>	
	(a)			(b)	

Figure 19.1: Motivating examples.

effect that a causal property (e.g., a datarace) in one model might not be recognized as such by another model.

19.1.1 Motivating Examples

Each example in Figure 19.1 shows a two-threaded program, together with one of its possible executions, in which Thread 1 is executed completely before Thread 2 starts. In this representation of executions, synchronized blocks are boxed, while write and read operations on shared locations are denoted by \leftarrow (receiving a value), and \rightarrow (yielding a value), respectively. Both *programs* exhibit a race condition between the two write operations on *y*. However, are the observed executions also exhibiting a causal datarace?

When analyzing the observed execution in Figure 19.1(a), a simple happens-before approach ordering all accesses to concurrent objects [236] cannot observe a causal datarace: the release operation of the lock in Thread 1 is required to happen-before the acquire of the lock in Thread 2. Happens-before with lock atomicity [207] is not able to infer a causal datarace either: although the lock atomicity would allow for the two lock-blocks to be permuted, the read of *x* in Thread 1 is still required to happen-before the write of *x* in Thread 2. Yet, the race condition can be captured as a causal datarace of the observed execution by weaker happens-before models [243, 267, 246], since in those models, one can additionally permute a write before a read of same location, as long as it is permuted before the write

corresponding to that read. Thus, the trace generated by the program in Figure 19.1(a) *has or does not have* a causal datarace, depending upon the particular causal model employed.

However, none of the approaches mentioned above can detect the race condition in Figure 19.1(b) as a causal datarace for the observed execution. The reason for this is that all models enforce at least the read-after-write dependency (i.e., a read should always follow the latest write event of the same variable), and therefore would not allow the permutation of the last two lock-blocks of the execution, since the read of x in Thread 2 must follow the last write of x in Thread 1. Nevertheless, there is enough information in the observed execution to be able to detect the race: since both writes of x in Thread 1 write the same value, it is actually possible to permute the last two lock blocks, and thus detect the race. Moreover, since one could conceive a technique specialized for finding such cases, it can be rightfully claimed that the observed execution has in fact a causal datarace, although not captured by any existing definition.

Given this ever increasing (regarding coverage) sequence of causal models and definitions for causal properties, it is only natural to ask the following question:

Is there any causal model that generalizes all existing models, and which cannot be surpassed?

We answer this question positively in the context of sequential consistency [179]. While we believe the presented approach can be applied to other memory models, we chose sequential consistency here for two reasons: (1) it is broadly accepted, popular and intuitive; (2) it is subsumed by other memory models: errors detected under sequential consistency are also errors for other memory models.

Contributions. The main result of this paper is a semantic framework that allows to prove maximality of causal models, and a proof that our proposed model is indeed the maximal causal model for the observed execution. This means that it comprises precisely *all* traces which can be generated by all programs which can generate the observed trace. Concretely, we show that: (1) all programs which can produce the observed execution can generate all traces in the model; and (2) for any trace not in the model there exists a program generating the observed trace which cannot generate it. To our knowledge, this is the first such result for causal models. We then prove

(the implicitly assumed) soundness for a series of existing causal models by showing they are submodels of the proposed model.

Paper structure. Section 19.2 introduces some notation and discusses sequential consistency. Section 19.3 axiomatizes consistent concurrent systems and defines our proposed causal models. Section 19.4 formally defines the maximality claim and proves our model maximal among sound models. Section 19.5 shows how existing models are included in ours, thus proving their soundness. Section 19.8 reviews related research and discusses several research ideas connected with the presented work. Section 19.9 concludes.

19.2 Execution Model

Assume a machine that can execute arbitrarily many threads in parallel. The execution environment contains a set of *concurrent objects* (shared memory locations, locks, ...), which are accessed by threads to share data and synchronize. *Threads*, which can only interact through the execution environment, are abstracted as *sequences of operations on concurrent objects*. The only source of thread non-determinism is the execution environment, that is, if the interaction between a thread and the environment is the same across executions, the thread will execute the same operations, in the same order. To simplify the presentation, we assume no dynamic creation of threads (this presents no technical difficulty).

19.2.1 Concurrent Objects, Serial Specification

We adopt the definition of concurrent objects and serial specifications proposed by Herlihy and Wing [142]. A concurrent object is behaviorally defined through a set of atomic operations, which any thread can perform on it, and a serial specification of its legal behavior in isolation. The serial specification describes the valid sequences of operations which can be performed on the object. We next describe two common types of concurrent objects.

Shared memory locations Each shared memory location can be regarded as a shared object with read and write operations, whose serial specification states that each read yields the same value as the one of the previous write. Moreover, to avoid non-determinism due to the initial state of

the memory, we will further require that all memory locations are initialized, that is, the first operation for each location is a write.

Mutexes Each mutex can be regarded as a concurrent object providing *acquire* and *release* operations. Their mutual exclusion property is achieved through the serial specification which accepts only those sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for each prefix, and all consecutive pairs of *acquire-release* share the same thread.

To keep the proofs simple and the concepts clear, we refrain here from adding more concurrency constructs (such as spawn/join, wait/notify, or semaphores). Note, however, that this would not introduce additional complexity, but just further constrain the notion of consistency.

19.2.2 Events and Traces

Operations performed by threads on concurrent objects are recorded as *events*. We consider events to be abstract entities from an infinite “collection” **Events**, and describe them as tuples of *attribute-value* pairs. The only attributes considered here are: *thread*—the unique id of the thread generating the event, *op*—the operation performed (e.g., *write*, *read*, *acquire*, or *release*), *target*—the concurrent object accessed by the event, and *data*—the value sent/received by the current event, if such exists (e.g., for the *write/read* operations). For example, $(thread=t_1, op=write, target=x, data=1)$ describes an event recording a write operation by thread t_1 to memory location x with value 1. When there is no confusion, we only list the attribute values in an event, e.g., $(t_1, write, x, 1)$. Our choice for deciding what attributes to record in an event considers a monitor which can observe memory and synchronization operations and the identity of the thread performing them, but has no access to the actual code. Section 19.8 includes a discussion on possible variations on the set of attributes recorded for an event.

For any event e and attribute $attr$, $attr(e)$ denotes the value corresponding to the attribute $attr$ in e , and $e[v/attr]$ to denote the event obtained from e by replacing the value of attribute $attr$ by v . An *execution trace* is abstracted as a sequence of events. Given a trace τ , a concurrent object o and a thread t , let $\tau|_o$ and $\tau|_t$ denote the restriction of τ to events involving only o , and only t , respectively. Let $latest_o(\tau)$ be the latest event of τ having the *op* attribute o . If o is omitted, it simply means the latest event in τ .

Sequential consistency can be now elegantly defined:

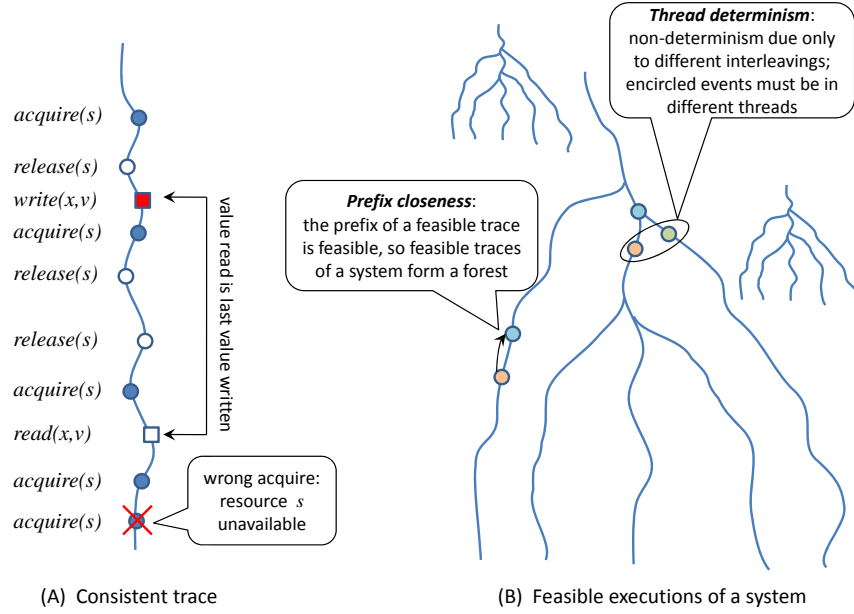


Figure 19.2: Feasibility model.

Definition 111 ([21]) *Let τ be any trace.*

- (1) τ is **legal** if and only if $\tau|_o$ satisfies o 's serial specification for any object o ;
- (2) An **interleaving** of τ is a trace τ' such that $\tau'|_t = \tau|_t$ for each thread t .
- (3) A trace τ is **(sequentially) consistent** if it admits a legal interleaving.

Since we restrict ourselves to sequential consistency, from here on when we say that a trace is sequentially consistent we automatically mean that it is also legal.

19.3 Feasibility Model

This section introduces an axiomatization for a machine producing consistent traces, and uses it to associate a sound-by-definition causal model to any observed execution, comprising all executions which can potentially be inferred from that execution alone, without additional knowledge of the system generating it.

Figure 19.2 highlights the two major concepts underlying our approach,

namely *trace consistency* and *feasible executions*. A consistent trace (Definition 111) disallows “wrong” behaviors, such as reading a value different from the one which was written, or proceeding when a lock cannot be acquired. Feasible executions, defined below, refer to *sets* of execution traces and aim at capturing *all* the behaviors that a given system or program can manifest. No matter what task a concurrent system or program accomplishes, its set of traces must obey some basic properties. First, feasible traces are *generatable*, meaning that any prefix of any feasible trace is also feasible; this is captured by our first axiom of feasible traces, *prefix closedness*. Second, we assume that thread interleaving is the only source of non-determinism in producing traces; this is captured by our second axiom of feasible traces, *local determinism*.

Each particular multithreaded system or programming environment, say \mathcal{S} , has its own notion of feasible execution, given by its specific intended semantics. Let us call all (possibly incomplete) traces that \mathcal{S} can yield \mathcal{S} -*feasible*, and let $feasible(\mathcal{S})$ be their set. Instead of defining $feasible(\mathcal{S})$, which requires a formal definition of \mathcal{S} and is therefore \mathcal{S} -specific (and tedious), we here *axiomatize* it:

Prefix Closedness

Events are indivisible and generated in execution order; hence, $feasible(\mathcal{S})$ must be prefix closed: if $\tau_1\tau_2$ is \mathcal{S} -feasible, then τ_1 is \mathcal{S} -feasible. Prefix closedness ensures that each event is generated individually, with the possibility of interleaving happening in-between any of them. For example, although the `++ x` instruction generates two events, a read, follow by a write on `x` these are not necessarily consecutive: if the instruction is not properly synchronized, another thread could write `x` after the first event, yielding an atomicity violation.

Local Determinism

The execution of a concurrent operation is determined by the previous events in the same thread, and can happen at any consistent moment after them. Formally, if $\tau e, \tau' \in feasible(\mathcal{S})$ and $\tau \upharpoonright_{thread(e)} = \tau' \upharpoonright_{thread(e)}$ then: if $\tau' e$ is consistent then $\tau' e \in feasible(\mathcal{S})$; moreover, if $op(e) = read$ and there exists an event e' such that $e = e'[data(e)/data]$ and $\tau' e'$ is consistent, then $\tau' e' \in feasible(\mathcal{S})$.

Note that there was a crucial typo in the previous version of this text and in [244]: $\tau'e$ instead of $\tau'e'$

The second part says that if a *read* operation is enabled, i.e., all previous events have been generated, then it can be executed at any consistent time (despite the fact that the value it receives might be different from that observed in the original trace). Allowing traces where read events observe a different value than in the original trace might seem like a source of unsoundness. Note, however, that the same local determinism property prohibits the thread on which such a read event occurred to continue after producing this event, by stating that an additional event for a thread is generated *only* if the current trace for that thread is *exactly* the same (including the value) as in the original trace. Suppose, for example, that two threads, identified by t_1 and t_2 , assign 1, then execute an increment operation on the same location l . One potential observed trace could be:

$(t_1, \text{write}, l, 1)(t_2, \text{write}, l, 1)(t_1, \text{read}, l, 1)(t_1, \text{write}, l, 2)(t_2, \text{read}, l, 2)(t_2, \text{write}, l, 3)$.
Local determinism ensures that we can also obtain the (partial) trace

$$(t_1, \text{write}, l, 1)(t_2, \text{write}, l, 1)(t_1, \text{read}, l, 1)(t_2, \text{read}, l, 2)(t_1, \text{write}, l, 2).$$

This shows that we can use local determinism to interleave threads differently than their original scheduling, as long as consistency is respected and threads produce the same events. Note that, (1) event $e = (t_2, \text{read}, l, 2)$ can be generated although it reads a different value than it originally did; and (2) thread t_1 can continue after e was generated (since it concerns a different thread), but thread t_2 cannot (because, e.g., e could be guarding a control statement).

Definition 112 \mathcal{S} is **consistent** iff $\text{feasible}(\mathcal{S})$ satisfies the axioms above.

A major goal of trace-based analysis is to infer/analyze as many traces as possible using a recorded trace. When one does not know (or does not want to use) the source code of the multithreaded program being executed, one can only infer potential traces of the system resembling the observed trace. Let us now define the proposed causal model, termed *feasibility closure*, as the set of executions which can be inferred from an observed execution—they correspond to the traces obtainable from τ using the feasibility axioms.

Definition 113 The **feasibility closure** of a consistent trace τ , written $\text{feasible}(\tau)$, is the smallest set of traces containing τ which is prefix-closed

and satisfies the local determinism property. A trace in $\text{feasible}(\tau)$ is called τ -feasible.

Without dwelling into details here, as this is proved elsewhere [79], intuitively the feasibility closure of a trace contains all interleavings of the observed trace, where each thread is stopped once it read a value from the memory different from the one observed originally, as well as all prefixes of these traces.

Add details from [79]

The following result formalizes the soundness of the proposed model. Assuming the base axioms are sound, the closure properties guarantee that all traces in our causal model are feasible. In addition, Proposition 48 shows that *any* system/program which can generate one trace, can also generate *all* traces comprised by its causal model.

Proposition 48 *If \mathcal{S} consistent and $\tau \in \text{feasible}(\mathcal{S})$ then $\text{feasible}(\tau) \subseteq \text{feasible}(\mathcal{S})$. Moreover, if τ' is consistent and $\tau \in \text{feasible}(\tau')$, then $\text{feasible}(\tau) \subseteq \text{feasible}(\tau')$.*

Proof: The intuition for $\tau \in \text{feasible}(\tau')$ is that if a run of any program executed on \mathcal{S} can produce τ' , then there is also some run of the same program executed also on \mathcal{S} that can produce τ . Specifically, both $\text{feasible}(\mathcal{S})$ and $\text{feasible}(\tau')$ are closed under the feasibility axioms. Since τ belongs to both of them, and $\text{feasible}(\tau)$ is the smallest set closed under the same axioms, it follows that it must be included in both. \square

19.4 Maximality

In this section we show that the proposed causal model is *maximal* among sound models, in the sense that any extension to it is done at the expense of soundness. We will prove therefore that given a trace τ' which is not in the feasibility closure of a trace τ , there exists a program p which can generate τ but not τ' ; therefore, if the model were extended to include τ' and used τ' as a witness that a property is satisfied/invalidated by a program generating τ , this would be a false witness if the program which generated τ was p .

To prove our claim, we propose CONC, a very simple (not even Turing complete) concurrent language. The benefit of such a simple language is that

CONC SYNTAX:	$ \begin{aligned} Proc &::= Proc \parallel Proc \mid Int : Stmt \\ Stmt &::= Stmt ; Stmt \mid \text{nop} \mid \text{if } Int \text{ then } Stmt \\ &\mid \text{load } Loc \mid Loc := Int \mid \text{acquire } Loc \mid \text{release } Loc \end{aligned} $	
CONC SEMANTICS:	$ \frac{\langle p_1, \sigma, \delta, \rho \rangle \xrightarrow{\tau} \langle p'_1, \sigma', \delta', \rho' \rangle}{\langle p_1 \parallel p_2, \sigma, \delta, \rho \rangle \xrightarrow{\tau} \langle p'_1 \parallel p_2, \sigma', \delta', \rho' \rangle} \quad (Par_1) $	
	$ \frac{\langle p_2, \sigma, \delta, \rho \rangle \xrightarrow{\tau} \langle p'_2, \sigma', \delta', \rho' \rangle}{\langle p_1 \parallel p_2, \sigma, \delta, \rho \rangle \xrightarrow{\tau} \langle p_1 \parallel p'_2, \sigma', \delta', \rho' \rangle} \quad (Par_2) $	
	$ \frac{\langle s, \sigma, \delta, \rho, t \rangle \xrightarrow{\tau} \langle s', \sigma', \delta', \rho', t \rangle}{\langle t : s, \sigma, \delta, \rho \rangle \xrightarrow{\tau} \langle t : s', \sigma', \delta', \rho' \rangle} \quad (Thread) $	
	$ \frac{\langle s_1, \sigma', \delta', \rho', t \rangle \xrightarrow{\tau} \langle s'_1, \sigma', \delta', \rho', t \rangle}{\langle s_1 ; s_2, \sigma, \delta, \rho, t \rangle \xrightarrow{\tau} \langle s'_1 ; s_2, \sigma', \delta', \rho', t \rangle} \quad (Seq) $	
	$ \frac{\cdot}{\langle \text{nop} ; s, \sigma, \delta, \rho, t \rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t \rangle} \quad (Nop) $	
	$ \frac{\cdot}{\langle \text{if } i \text{ then } s, \sigma, \delta, \rho, t \rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t \rangle} \text{ if } \rho(t) = i \quad (If_{true}) $	
	$ \frac{\cdot}{\langle \text{if } i \text{ then } s, \sigma, \delta, \rho, t \rangle \xrightarrow{\epsilon} \langle \text{nop}, \sigma, \delta, \rho, t \rangle} \text{ if } \rho(t) \neq i \quad (If_{false}) $	
	$ \frac{\cdot}{\langle \text{load } x, \sigma, \delta, \rho, t \rangle \xrightarrow{(t, read, x, i)} \langle \text{nop}, \sigma, \delta, \rho[t \leftarrow i], t \rangle} \text{ where } i = \sigma(x) \quad (Read) $	
	$ \frac{\cdot}{\langle x := i, \sigma, \delta, \rho, t \rangle \xrightarrow{(t, write, x, i)} \langle \text{nop}, \sigma[x \leftarrow i], \delta, \rho, t \rangle} \quad (Write) $	
	$ \frac{\cdot}{\langle \text{acquire } x, \sigma, \delta, \rho, t \rangle \xrightarrow{(t, acquire, x)} \langle \text{nop}, \sigma, \delta[x \leftarrow t], \rho, t \rangle} \text{ if } \delta(x) = \perp \quad (Acq) $	
	$ \frac{\cdot}{\langle \text{release } x, \sigma, \delta, \rho, t \rangle \xrightarrow{(t, release, x)} \langle \text{nop}, \sigma, \delta[x \leftarrow \perp], \rho, t \rangle} \text{ if } \delta(x) = t \quad (Rel) $	

Figure 19.3: Syntax and SOS semantics for the CONC language

it can conceivably be simulated in any real language; therefore, proving the maximality result for CONC proves the model is maximal for all languages.

Figure 19.3 presents the grammar and SOS semantics of CONC. The grammar specifies a parallel composition of named threads. Each thread is a succession of statements and uses one internal register to load data from the shared memory. `load x` loads the value at location x into the internal register of the thread, `$x := i$` stores integer i at location x , `acquire` and `release` have the straight-forward semantics, and `if i then s` executes s only if the internal register has value i . A running configuration of CONC is a tuple $\langle p, \sigma, \delta, \rho \rangle$ where p is the remainder of the program being executed, σ maps variables to values, δ maps each lock to the id of the thread holding it, and ρ gives for each thread the value of its internal register. In the SOS derivation rules we additionally use configurations of the form $\langle p, \sigma, \delta, \rho, t \rangle$, where t is the thread id obtained in the (*Thread*) rule, which is propagated by all following rules. Assuming p has n threads, the initial configuration of the system is $START(p) = \langle p, \sigma_\epsilon, \delta_\epsilon, \rho_\epsilon^n \rangle$ where σ_ϵ , δ_ϵ , and ρ_ϵ^n initialize all locations, locks, and registers for the n threads with \perp , respectively.

We have chosen this minimal language both because it is sufficiently expressive to generate all (finite) legal traces, and because it is quite easy to mimic in any other language. In Java, for example, each thread would be modeled by a thread object, and all threads could be started in a loop by the main thread. Since beginnings of threads do not generate events, this is as-if all threads start together in parallel. The running method of each Java thread object would declare a local variable r to stand for the register, and then the two CONC instructions dealing with the register translate as follows: `load l` becomes $r = l$, and `if i then s` becomes `if ($r == i$) s` .

It is straightforward to associate to each event an instruction producing it. Let *code* be the mapping defined on events as follows:

$$code(e) = \begin{cases} \text{load } x & \text{if } e = (t, read, x, i) \\ x := i & \text{if } e = (t, write, x, i) \\ \text{acquire } x & \text{if } e = (t, acquire, x) \\ \text{release } x & \text{if } e = (t, release, x) \end{cases}$$

Given a program p , let $p|_t$ be its projection on thread t , that is, the statement labeled by t in the parallel composition.

The following result shows that, except for the code, the running configuration is completely determined by the trace generated up to that point:

Proposition 49 *If $\text{CONC} \vdash START(p) \xrightarrow{\tau}^* \langle p', \sigma_\tau, \delta_\tau, \rho_\tau^n \rangle$, where n is the number of threads of p , then:*

- (1) $\sigma_\tau(x) = \text{data}(\text{latest}_{\text{write}}(\tau \upharpoonright_x));$
- (2) $\delta_\tau(x) = \begin{cases} \text{thread}(\text{latest}(\tau \upharpoonright_x)), & \text{if } \text{op}(\text{latest}(\tau \upharpoonright_x)) = \text{acquire} \\ \perp, & \text{otherwise} \end{cases};$
- (3) $\rho_\tau^n(t) = \text{data}(\text{latest}_{\text{read}}(\tau \upharpoonright_t)).$

Proof: First note that ϵ -transitions only affect the code-part of configurations. Therefore, the base case, when $\tau = \epsilon$, is trivial. Suppose now that

$$\text{START}(p) \xrightarrow{\tau^*} \langle p', \sigma_\tau, \delta_\tau, \rho_\tau^n \rangle \xrightarrow{e} \langle p'', \sigma, \delta, \rho \rangle.$$

Suppose that $\text{thread}(e) = t_i$. Then in order for e to be generated, $p' \upharpoonright_{t_i}$ must be $\text{code}(e); p'''$ and $p'' = \text{nop}; p'''$.

The proof tree is built by applying in order $i - 1$ steps of Par_2 and then a step of (Par_1) (or none, if $i = n$), then (Thread) , (Seq) , and finally one of the (Read) , (Write) , (Acq) , or (Rel) ; we therefore need to show that

$$\langle \text{code}(e), \sigma_\tau, \delta_\tau, \rho_\tau^n, t_i \rangle \xrightarrow{e} \langle \text{nop}, \sigma_{\tau e}, \delta_{\tau e}, \rho_{\tau e}^n, t_i \rangle$$

If $e = (t_i, \text{read}, x, i)$, then $\text{code}(e) = \text{load } x$ so we can apply the (Read) rule, which updates only ρ_τ^n to $\rho_\tau^n[r \leftarrow i] = \rho_{\tau e}^n$.

If $e = (t_i, \text{write}, x, i)$, then $\text{code}(e) = x := i$, and rule Write applies updating only σ to $\sigma_\tau[x \leftarrow i] = \sigma_{\tau e}$.

If $e = (t_i, \text{acquire}, x)$, then $\text{code}(e) = \text{acquire } x$ and only δ is updated to $\delta_\tau[x \leftarrow t] = \delta_{\tau e}$.

If $e = (t_i, \text{release}, x)$, then $\text{code}(e) = \text{release } x$ and only δ is updated to $\delta_\tau[x \rightarrow \perp] = \delta_{\tau e}$. \square

Therefore, in the sequel we will use $\text{CONC} \vdash p \xrightarrow{\tau^*} p'$ instead of $\text{CONC} \vdash \text{START}(p) \xrightarrow{\tau^*} \langle p', \sigma_\tau, \delta_\tau, \rho_\tau^n \rangle$

Now, let us prove that the semantics of CONC does indeed satisfy the sequential consistency axioms. Let p be a CONC program and let $\text{feasible}(p)$ be the set of all p -feasible traces; that is τ is p -feasible if there exists a program p' such that $\text{CONC} \vdash p \xrightarrow{\tau^*} p'$. We will show that $\text{feasible}(p)$ satisfies the *strong local determinism* property, namely, not only that an enabled event can be generated at any point by its thread, but that it also must be the (unique) next event generated by that thread (ignoring the *data* attribute for *read* events). Formally,

Definition 114 *feasible*(p) satisfies the **strong local determinism** property if it satisfies local determinism and if $\tau_1 e_1$ and $\tau_2 e_2$ are p -feasible,

$thread(e_1) = thread(e_2) = t$, and $\tau_1 \upharpoonright_t = \tau_2 \upharpoonright_t$, then $op(e_1) = op(e_2)$, and $target(e_1) = target(e_2)$; if additionally $op(e_i) = write$, then also $data(e_1) = data(e_2)$.

The following result shows that every CONC program p is a consistent system in the sense of Definition 112.

Proposition 50 (CONC consistency) *feasible(p) satisfies prefix closedness and strong local determinism.*

Proof: [Sketch] Prefix closedness is obvious, since the semantics can emit at most one event for each execution step. The second property follows by case analysis on the rule SOS rule applied to produce the relevant event for local determinism. \square

Proof: Prefix closedness is obvious, since the semantics can emit at most one event for each execution step.

Now, notice that every transition in CONC modifies the code for precisely one of the threads. Let us prove the following stronger local result which basically shows that the evolution of a thread does not depend on events which are not directly related to it.

Suppose $CONC \vdash p \xrightarrow{\tau_1}^* p_1 \xrightarrow{\tau'_1} p'_1$ (where τ'_1 is either ϵ or an event) and $CONC \vdash p \xrightarrow{\tau_2}^* p_2$, such that $\tau_1 \upharpoonright_t = \tau_2 \upharpoonright_t$, $p_1 \upharpoonright_t = p_2 \upharpoonright_t$, $p'_1 \upharpoonright_t \neq p_1 \upharpoonright_t$, and either $\tau_2 \tau'_1$ is consistent or τ'_1 is a *read* event. Then there exist a unique p'_2 such that $p'_2 \upharpoonright_t \neq p_2 \upharpoonright_t$ and $CONC \vdash p \xrightarrow{\tau_2}^* p_2 \xrightarrow{\tau'_2} p'_2$, and, moreover $p'_1 \upharpoonright_t = p'_2 \upharpoonright_t$, and either $\tau'_1 = \tau'_2$ or both of them are reading from the same target but with different values.

Note that uniqueness holds trivially, as once a thread was chosen, at most one rule can apply. Now, for the existence part, if the transition for τ'_1 is

(*Nop*), then it can be applied on p_2 with the same effect;

(*If_{true}*) or (*If_{false}*), then since $\rho_\tau^n(t) = data(latest_{read}(\tau \upharpoonright_t))$ and $\tau_1 \upharpoonright_t = \tau_2 \upharpoonright_t$, it follows that $\rho_{\tau_1}^n(t) = \rho_{\tau_2}^n(t)$, therefore the exact same transition can be applied on p_2 ;

(*Read*), then $\tau'_1 = (t, read, x, i)$, which must be generated by an instruction load x , whence the same rule can be applied on p_2 , generating event $(t, read, x, i')$, where $i' = \sigma_{\tau_2}(x)$.

(*Write*), then $\tau'_1 = (t, \text{write}, x, i)$, which must be generated by an instruction $x := i$, whence and same rule can be applied on p_2 , generating the same event;

(*Acq*), then $\tau'_1 = (t, \text{acquire}, x)$, which must be generated by **acquire** x ; since $\tau_2\tau'_1$ is consistent, it must be that $\delta_{\tau_2}(x) = \perp$, hence the rule can be applied on p_2 generating the same event;

(*Rel*), then $\tau'_1 = (t, \text{release}, x)$, which must be generated by **release** x ; since $\tau_2\tau'_1$ is consistent, it must be that $\delta_{\tau_2}(x) = t$, hence the rule can be applied on p_2 generating the same event.

Let us now use the above lemma to prove the local determinism property. Assume $\tau_1 e$ and τ_2 are p -feasible traces such that $\text{thread}(e) = t$ and $\tau_1 \upharpoonright_t = \tau_2 \upharpoonright_t$. Let p_1, p_2 be such that $\text{CONC} \vdash p \xrightarrow{\tau_1 e} p_1$ and $\text{CONC} \vdash p \xrightarrow{\tau_2} p_2$. Then we can use the lemma proved above to show that $[\text{CONC} \vdash p \xrightarrow{\tau_2}^* p_2] \upharpoonright_t$ is a prefix of $[\text{CONC} \vdash p \xrightarrow{\tau_1 e}^* p_2] \upharpoonright_t$, by induction on the length of $[\text{CONC} \vdash p \xrightarrow{\tau_2}^* p_2] \upharpoonright_t$. Moreover, using the same lemma, we can (uniquely) continue the execution of p_2 on thread t using the same steps from the execution of p_1 and the fact that either $\tau_2 e$ is consistent or e is a read event.

For strong local determinism, if $\tau_1 e_1$ and $\tau_2 e_2$ are p -feasible, $\text{thread}(e_1) = \text{thread}(e_2) = t$, and $\tau_1 \upharpoonright_t = \tau_2 \upharpoonright_t$, then we can use an argument similar to the one above to match the statements corresponding to thread t from the beginning of the execution and until e_1 and e_2 are generated, and for this step, applying the part of the lemma referring to the generated events. \square

Now, given a trace τ , let us build the canonical CONC program generating it. *code* can be naturally extended on traces by $\text{code}(e\tau) = \text{code}(e) ; \text{code}(\tau)$. Let $\{t_1, t_2, \dots, t_n\}$ be the set of thread ids appearing in τ . Then the program associated to a trace τ is defined by $\text{program}(\tau) = t_1 : \text{code}(\tau \upharpoonright_{t_1}) \parallel \dots \parallel t_n : \text{code}(\tau \upharpoonright_{t_n})$.

Let us also define the empty program with n threads as $\text{program}^n(\epsilon) = t_1 : \text{nop} \parallel \dots \parallel t_n : \text{nop}$. The following result shows that the program corresponding to a consistent trace can indeed generate that trace.

Proposition 51 *If τ is a consistent trace with n threads, then $\text{CONC} \vdash \text{program}(\tau) \xrightarrow{\tau}^* \text{program}^n(\epsilon)$.*

Proof: We prove by induction on the length of τ that $\text{CONC} \vdash \text{program}(\tau\tau') \xrightarrow{\tau}^* \text{program}^n(\tau')$ where

$$\text{program}^n(\tau) = t_1 : \overline{\text{code}}(\tau \upharpoonright_{t_1}) \parallel \dots \parallel t_n : \overline{\text{code}}(\tau \upharpoonright_{t_n}), \quad \overline{\text{code}}(\tau) = \begin{cases} \text{code}(\tau) & \text{if } \tau \neq \epsilon \\ \text{nop} & \text{otherwise} \end{cases}.$$

The base case, when $\tau = \epsilon$, is trivial: $program^n(\tau') = program(\tau)$. Suppose now that $CONC \vdash program(\tau e \tau') \xrightarrow{\tau^*} program^n(e \tau')$. We need to show that

$$\langle program^n(e \tau'), \sigma_\tau, \delta_\tau, \rho_\tau^n \rangle \xrightarrow{e^*} \langle program^n(\tau'), \sigma_{\tau e}, \delta_{\tau e}, \rho_{\tau e}^n \rangle.$$

Suppose that $thread(e) = t_i$. Then:

$$program^n(e \tau') \upharpoonright_{t_j} = \begin{cases} program^n(\tau') \upharpoonright_{t_j}, & \text{if } j \neq i \\ code(e) ; program^n(\tau') \upharpoonright_{t_i}, & \text{otherwise} \end{cases}$$

We can build a proof for the above assertion in two execution steps: one generating the event and affecting the configuration by turning the instruction $code(e)$ into **nop**, and the other dissolving the **nop** and obtaining the desired configuration. For the first step, the proof tree is build by applying in order $i - 1$ steps of Par_2 and then a step of (Par_1) (or none, if $i = n$), then $(Thread)$, (Seq) , and finally one of the $(Read)$, $(Write)$, (Acq) , or (Rel) to deduce

$$\langle code(e), \sigma_\tau, \delta_\tau, \rho_\tau^n, t \rangle \xrightarrow{e} \langle \text{nop}, \sigma_{\tau e}, \delta_{\tau e}, \rho_{\tau e}^n, t \rangle$$

If $e = (t, read, x, i)$, then $code(e) = \text{load } x$ so we can apply the $(Read)$ rule, which updates the register for t of ρ_τ^n to $\sigma_\tau(x) = data(latest_{write}(\tau \upharpoonright_x))$. However, due to consistency constraints, $data(e) = data(latest_{write}(\tau \upharpoonright_x))$, whence $\rho_\tau^n[r \leftarrow i] = \rho_{\tau e}^n$ and the rule generates e . Moreover, $\sigma_{\tau e} = \sigma_\tau$ and $\delta_{\tau e} = \delta_\tau$, whence our claim is proven.

If $e = (t, write, x, i)$, then $code(e) = x := i$, and rule $Write$ applies generating e ; we have that $\sigma_\tau[x \leftarrow i] = \sigma_{\tau e}$, $\delta_{\tau e} = \delta_\tau$, and $\rho_{\tau e}^n = \rho_\tau^n$.

If $e = (t, acquire, x)$, then $code(e) = \text{acquire } x$ and because of the consistency requirements for mutexes (the difference between the number of *acquire* and *release* operation is either 0 or 1 for each prefix), it must be that $op(latest(\tau \upharpoonright_x)) \neq \text{acquire}$, whence $\delta_\tau(x) = \perp$ and rule Acq can apply generating e ; we have that $\delta_\tau[x \leftarrow t] = \delta_{\tau e}$, $\sigma_\tau = \sigma_{\tau e}$, and $\rho_\tau^n = \rho_{\tau e}^n$.

If $e = (t, release, x)$, then $code(e) = \text{release } x$ and because of the consistency requirements for mutexes, it must be that $op(latest(\tau \upharpoonright_x)) = \text{acquire}$, whence $\delta_\tau(x) = thread(latest(\tau \upharpoonright_x))$. Again, from consistency requirements, since all consecutive *acquire-release* pairs share the same thread, it follows that $\delta_\tau(x) = t$ and so the rule Rel can apply generating e ; we have that $\delta_\tau[x \leftarrow \perp] = \delta_{\tau e}$, $\sigma_\tau = \sigma_{\tau e}$, and $\rho_\tau^n = \rho_{\tau e}^n$. \square

The following theorem justifies the maximality claims for the proposed model.

Theorem 28 (Maximality) *For any consistent trace τ' which is not τ -feasible there exists a program generating τ but not τ' .*

Proof: [Sketch] Because of prefix closeness and thread determinism, the only interesting case to analyze is when τ' continues the execution on a thread after reading a value distinct from the one recorded in an event e of τ . In that case, we create a new program p from $program(\tau)$ by inserting a conditional write instruction right after that generating event e . We then show that program p can still generate τ , but cannot generate τ' . \square

Proof: We can assume, without loss of generality, that $\tau' = \tau_1 e'$ such that τ_1 is τ -feasible (potentially empty).

Let $t = thread(e')$.

(1) Suppose $\tau_1 \upharpoonright_t$ is a prefix of $\tau \upharpoonright_t$. If $\tau \upharpoonright_t = \tau_1 \upharpoonright_t$ then τ' cannot be $program(\tau)$ -feasible because that would mean there exists a derivation $CONC \vdash program(\tau) \xrightarrow{\tau_1}^* p_1 \xrightarrow{e'} p'_1$; however since $CONC \vdash program(\tau) \xrightarrow{\tau}^* p$ with $p = program^n(\epsilon)$, from the proof of Proposition 50, $[CONC \vdash program(\tau) \xrightarrow{\tau}^* p] \upharpoonright_t$ must be a proper prefix of $[CONC \vdash program(\tau) \xrightarrow{\tau_1}^* p_1 \xrightarrow{e'} p'_1] \upharpoonright_t$, which is not possible as $p \upharpoonright_t = \text{nop}$.

Otherwise it must be that $\tau \upharpoonright_t = \tau_1 \upharpoonright_t e \tau_2$, and we can apply the strong local determinism to deduce that if τ' is $program(\tau)$ -feasible, then either $e = e'$ or their type is *read* and their states are different. But both these lead to contradiction because they imply that $\tau_1 e' \in feasible(\tau)$.

(2) If $\tau_1 \upharpoonright_t = \tau_0 e'_0$ is not a prefix of $\tau \upharpoonright_t$, then, because τ_1 is τ -feasible, it must be that $\tau \upharpoonright_t = \tau_0 e_0 e \tau_2$ such that $op(e_0) = op(e'_0) = \text{read}$, $target(e_0) = target(e'_0)$, and $data(e_0) \neq state(e'_0)$.

Let us consider a special event $?^t = (t, ?)$ (with the meaning that $thread(?^t) = t$ and $op(?^t) = ?$) and for each statement s , an extension $code^s$ of $code$ with $code^s(?^t) = s$, and $program^s(\tau) = code^s(\tau \upharpoonright_{t_1}) \parallel \dots \parallel code^s(\tau \upharpoonright_{t_n})$.

Let $x = target(e_0)$, $i = data(e_0)$ and let

$$j = \begin{cases} data(e') - 1, & \text{if } data(e') \text{ defined} \\ 0, & \text{otherwise} \end{cases}$$

Let τ_l, τ_r be such that $\tau = \tau_l e \tau_r$ and $\tau_l \upharpoonright_t = \tau_0 e_0$, and let $p' = program^s(\tau_l ?^t e \tau_r)$, obtained by inserting $s = \text{if } i \text{ then } x := j$ in the code for thread t , between the code for e_0 and that for e . This new program can still generate τ , as the state read by e_0 is different than that read by e'_0 and thus the conditional is not executed, but it cannot generate τ' , as the next event for thread t upon generating $\tau_0 e'_0$ must be $(t, write, x, j)$ which is guaranteed to be distinct from e' . \square

19.5 Proving Soundness of Existing Causal Models

Focusing on identifying concurrency anomalies and measuring success based on the number of bugs found, almost no causal model in the literature is actually proved sound. The authors of a causal model usually give some common-sense arguments for their choice and informally rely on the soundness of Happens-Before [179]. However, intuition can sometimes be misleading: in Section 19.5.4 we reveal a soundness problem with the model of Sen et al. [243]. Moreover, even when proved sound, the proofs are quite laborious, each having to repeat the formalization of an execution model. Proving soundness of other causal models by embedding them in our already proven sound model eliminates the need for an execution model and reduces proofs to checking closure properties.

We start with the following result, which can be regarded as a sufficient criterion for feasibility:

Theorem 29 *Any consistent prefix of an interleaving of τ is τ -feasible.*

Proof: Induction on the length of the interleaving prefix. The base case is trivial. Let $\tau'e$ be a consistent interleaving prefix of τ , and assume that τ' is τ -feasible. Let $t = \text{thread}(e)$, and let τ_1, τ_2 be such that $\tau = \tau_1 e \tau_2$. By prefix closedness, it follows that $\tau_1 e$ is feasible. Moreover, since $(\tau'e)|_t = \tau'|_t e$ is a prefix of $\tau|_t$, it follows that $\tau'|_t = \tau_1|_t$. Using the local determinism for $\tau_1 e$ and τ' , we obtain that $\tau'e$ is τ -feasible (since it is consistent). \square

The remainder of this section shows that existing sound causal models are captured by the feasibility closure as simple instances of Theorem 29. Another important consequence of Theorem 29 is that it basically shows there is a unique feasibility closure associated to a concurrent computation, regardless of the representative trace [79].

19.5.1 Happens Before Relation on Mazurkiewicz Traces

One elegant way to capture the happens-before trace equivalence is the Mazurkiewicz trace [194] associated to the dependence given by the happens-before relation.

The happens-before dependence is a set $T \cup D$, where $T = \bigcup_t \{(e_1, e_2) : \tau|_t = \tau_1 e_1 e_2 \tau_2\}$ is the intra-thread sequential dependence relation and $D = \bigcup_x \{(e_1, e_2) : \tau|_x = \tau_1 e_1 e_2 \tau_2 \text{ such that } e_1 \text{ or } e_2 \text{ is a write of } x\}$ is the sequential

memory dependence relation. Given this happens-before dependence, the Mazurkiewicz trace associated with τ is defined as the least set $[\tau]$ of traces containing τ and being closed under permutation of consecutive independent events [194]: if $\tau_1 e_1 e_2 \tau_2 \in [\tau]$ and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2 \in [\tau]$.

The following result shows that the feasibility closure is closed under the equivalence relation generated by happens-before, that is, happens-before is captured by our model, and thus re-shown sound for consistent executions:

Proposition 52 *If $\tau_1 e_1 e_2 \tau_2$ is τ -feasible and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2$ is τ -feasible. Given any τ -feasible trace τ' , $[\tau'] \subseteq \text{feasible}(\tau)$. Hence, $[\tau] \subseteq \text{feasible}(\tau)$.*

Proof: Let $\tau_1 e_1 e_2 \tau_2$ be a τ -feasible trace such that $(e_1, e_2) \notin T \cup D$. We will show that $\tau_1 e_2 e_1 \tau_2$ is also τ -feasible. First, all prefixes of $\tau_1 e_1 e_2 \tau_2$, including τ_1 , $\tau_1 e_1$, $\tau_1 e_1 e_2$, $\tau_1 e_1 e_2 \tau'_2 e'_2$ (for any prefix $\tau'_2 e'_2$ of τ_2), are τ -feasible, since $\text{feasible}(\tau)$ is prefix closed. Now, we can iteratively use closedness under local determinism (1) for $\tau_1 e_1 e_2$ and τ_1 , to derive that $\tau_1 e_2$ is τ -feasible; (2) for $\tau_1 e_1$ and $\tau_1 e_2$ to derive that $\tau_1 e_2 e_1$ is also τ -feasible; (3) by finitary induction for each prefix $\tau'_2 e'_2$ of τ_2 , for $\tau_1 e_1 e_2 \tau'_2 e'_2$ and $\tau_1 e_2 e_1 \tau'_2$ to derive that $\tau_1 e_2 e_1 \tau'_2 e'_2$ is also τ -feasible.

Therefore, for any τ -feasible trace τ' , $\text{feasible}(\tau')$ is closed under permutation of consecutive independent events; hence, $[\tau'] \subseteq \text{feasible}(\tau') \subseteq \text{feasible}(\tau)$. \square

19.5.2 Weak Happens Before

Several more recent trace analysis techniques [267, 243, 246] argue that the happens-before model can be further relaxed, noticing that the only purpose of the write-after-read happens-before order is to guarantee that a read event always reads the same write event as before in any feasible interleaving of the original trace. Therefore, one only needs to preserve the *read-after-write dependence*:

Definition 115 *Suppose $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$. Then e_2 **write-read depends on** e_1 in τ , written $e_1 <_{\tau}^{wr} e_2$, if $\text{target}(e_1) = \text{target}(e_2)$, $\text{op}(e_1) = \text{write}$, $\text{op}(e_2) = \text{read}$, and for all $e \in \mathcal{E}_{\tau_2}$, either $\text{target}(e) \neq \text{target}(e_1)$, or $\text{op}(e) \neq \text{write}$.*

That is, $e_1 <_{\tau}^{wr} e_2$ iff the value read by e_2 is the value written by e_1 .

Sen et al. [243] introduce the notion of *atomic* sets associated to each *write* event, containing itself and all read events which write-read depend

on it, accepting as feasible executions all linearizations of the transitive closure of the combined $<_{\tau}^{wr}$ and thread ordering, satisfying the additional requirement that the atomic sets are preserved. However this can be simply restated as follows [267]:

Definition 116 $\tau \sim \tau'$ if τ is an interleaving of τ' and $<_{\tau}^{wr} = <_{\tau'}^{wr}$.

That is, the \sim -equivalence class of τ contains all interleavings of τ which have exactly the same write-read dependence relation. Next result shows that this model is also captured by our model.

Proposition 53 If τ_1 is τ -feasible, and $\tau_1 \sim \tau_2$, then τ_2 is also τ -feasible.

Proof: We show that we are in the conditions of Theorem 29: Since τ_1 is consistent, and $<_{\tau_2}^{wr} = <_{\tau_1}^{wr}$, it follows that τ_2 must also be consistent, since all *read* events follow the same *write* events as in the τ_1 , which, by the consistency of τ_1 , precisely implies that each *read* event returns the value of the previous *write* event. \square

19.5.3 Happens-Before with synchronization

A conservative and sound approach, requiring no implementation changes, to handle locks in happens-before-based trace analysis techniques is to assume that *acquire* and *release* operations on the same lock yield the same happens-before dependence as if they were particular *write* and *read* operations (on the lock variable) [236]. However, this prevents synchronized blocks from being permuted, and thus imposes coverage limitations. The lock-set approaches, also called hybrid happens-before [207], propose to handle locks separately, associating with each event the set of locks [232] protecting them, hereby not enforcing any particular order between synchronized blocks.

We here group the events protected by locks in *atomic blocks*. Events e_1 and e_2 from a consistent trace τ , both generated by thread t , are *l-atomic* in τ , written $e_1 \Downarrow_l^{\tau} e_2$, if and only if there is some *acquire* event e on lock l generated by t before both e_1 and e_2 , and there is no *release* event e' on l generated by t between e and either of e_1, e_2 . For each lock l , let $[e]_l$ denote the *l-atomic equivalence class* of e . Assuming a trace in which all acquired locks are eventually released, *l-atomic* equivalence classes consist of all events belonging to the same acquire-release block of l . A trace τ' is *consistent with the lock atomicity* of τ if there exists no lock l and decomposition $\tau_1 e_1 \tau_2 e_2 \tau_3 e_3 \tau_4 e_4 \tau_5$ such that $e_1 \Downarrow_l^{\tau} e_3$ and $e_2 \Downarrow_l^{\tau} e_4$ and $[e_1]_l \neq [e_2]_l$. Let \prec_{hb}^{τ}

be the transitive closure of the union between happens-before and thread orderings of τ . The following holds:

Proposition 54 *Let τ' be a τ -feasible trace. Any linearization of $\prec_{hb}^{\tau'}$ consistent with the lock atomicity of τ' is τ -feasible.*

Proof: Again, we reduce our proof to Theorem 29. First, any linearization of $\prec_{hb}^{\tau'}$ is an interleaving of τ' . Moreover, since τ' is consistent, preservation of happens-before ensures that the serial specification of the memory locations is satisfied. Finally, consistency with lock atomicity implies that the serial specification for mutexes is also satisfied. Therefore, any linearization of $\prec_{hb}^{\tau'}$ consistent with the lock atomicity of τ' , is a consistent interleaving of τ' , thus τ' -feasible. \square

19.5.4 Weak-Happens-Before with synchronization

We next present two approaches to handling synchronization in weak-happens-before models and show they are both embeddable in our model.

Lock atomicity via write-read atomicity [243].

Since the notion of write-read atomicity already allows atomic sets to be permuted, it seems reasonable to use the conservative idea from standard happens-before methods, and treat *acquire* as a *write* event and *release* as a read event. Formally, given the consistent trace τ , one could additionally introduce an atomic dependence relation $<_{\tau}^a$ given by $e_1 <_{\tau}^a e_2$ if $\tau = \tau_1 e_2 \tau_2 e_1 \tau_3$, $target(e_1) = target(e_2)$, $op(e_1) = acquire$, $op(e_2) = release$, and there is no event e in τ_2 such that $target(e) = target(e_1)$, and $op(e) = acquire$. With this definition, equivalent traces to an observed trace τ are those interleavings of τ having the same write-read and atomic dependencies.

However, this definition needs a careful approach. Consider the example in Figure 19.1(b), and suppose that we observe a similar execution, but that the program is stopped after the *read* of x in Thread 2. Since no *release* event has been generated, the *acquire* in Thread 2 has no event depending on it, and thus it can be permuted (without the *read* event on x it was supposed to protect) before the last lock-block of Thread 1. Then, the final *read* of x itself can be permuted past the final *release* of l in Thread 1, exhibiting a spurious causal datarace.

Nevertheless, these models are sound for *synchronization complete* traces, that is, traces in which each acquired lock is eventually released.

Proposition 55 *Let τ_1 be a synchronization complete τ -feasible trace. Any interleaving τ_2 of τ_1 satisfying that $<_{\tau_2}^{wr} = <_{\tau_1}^{wr}$ and $<_{\tau_2}^a = <_{\tau_1}^a$ is τ -feasible.*

Proof: Since we already shown that $<_{\tau_2}^{wr} = <_{\tau_1}^{wr}$ implies that the serial specification of memory locations is verified, we only need to show that $<_{\tau_2}^a = <_{\tau_1}^a$ implies the satisfaction of the mutex specification for synchronization complete traces, that is, that any prefix of τ_2 has at most one more *acquire* operations than *release* operations, and all consecutive pairs of *acquire-release* have the same thread. The second part is easily guaranteed by the fact that $<_{\tau_2}^a = <_{\tau_1}^a$, since $<^a$ enforces the *acquire-release* in relation are consecutive, and, since τ_1 is consistent, this definition additionally implies that they have the same thread. The first part comes from the fact that, since τ_1 is synchronization complete, every *acquire* has a corresponding *release*, with whom it is in the $<_{\tau_1}^a$ relation. \square

Lock atomicity via locksets.

Wang and Stoller [267] propose a weak-happens-before model based on write-read dependence, while using locksets to handle locks as individual objects. In this model, a trace τ' is equivalent with a consistent trace τ if τ' is an interleaving of τ having the same write-read dependence relation and being consistent with the lock atomicity of τ .

Proposition 56 *Let τ_1 be a τ -feasible trace. Any interleaving τ_2 of τ_1 , consistent with the lock atomicity of τ_1 and satisfying that $<_{\tau_2}^{wr} = <_{\tau_1}^{wr}$ is τ -feasible.*

Proof: From the proof of Proposition 53, $<_{\tau_2}^{wr} = <_{\tau_1}^{wr}$ implies the serial specification of memory locations is obeyed in τ_2 . Additionally, from the proof of Proposition 54, consistency with the lock atomicity of a consistent trace implies that the serial specification of mutexes is obeyed. We can therefore apply Theorem 29. \square

19.6 Characterizing the Feasibility Closure

Section 19.3 showed how our causal model can be naturally defined by characterizing feasibility axiomatically rather than constructively. Closure axioms guarantee that all equivalent traces which can be derived based on the consistency axioms are considered. This section presents a constructive characterization of the feasibility closure.

As might have been suggested by Theorem 29, consistent interleaving prefixes cover all possibilities of generating τ -feasible traces using only the events in τ . However, the definition of interleaving (prefix) overlooks the final part of the local determinism axiom, that is, the one regarding operations which might receive different values from their objects. To achieve a complete constructive characterization of the feasibility closure, we have to go beyond prefixes of interleavings, more exactly, one *read* operation per thread beyond. This is because, as guaranteed by local determinism, whenever all events before an event have been generated in a thread, the operation on the concurrent object specified in that event can also take place, but its *state* attribute might now retrieve a value different from the one it had in the observed trace. However, once such an event whose *state* is different from the one in the original trace is derived, the execution cannot be continued for that thread, because that event might influence/prevent the generation of the following events. An *extended interleaving prefix* is a (partial) trace which behaves similarly to the observed trace up to its final event for each thread, which might have a different value:

Definition 117 *Trace $\tau' = \tau_1 e'$ is an **extended prefix** of $\tau = \tau_1 e \tau_2$ if either $e = e'$, or $op(e) = read$ and $e = e'[state(e)/state]$. τ' is an **extended interleaving prefix** of τ if $\tau'|_t$ is an extended prefix of $\tau|_t$ for any thread t .*

We can now give a complete constructive characterization for τ -feasible traces:

Theorem 30 *Given τ consistent, a trace τ' is τ -feasible iff it is a consistent extended interleaving prefix of τ .*

Proof: Proving that any consistent extended interleaving prefix of τ is τ -feasible proceeds similarly to the proof of Theorem 29. For the reverse, one needs to show that the set of consistent extended interleaving prefixes of τ contains τ , is prefix closed, and closed under local determinism. First two are obvious: τ is an interleaving prefix of itself, and any prefix of an extended interleaving prefix of τ is an extended interleaving prefix of τ by the definition. Now let $\tau_1 e$ and τ_2 be consistent interleaving prefixes of τ such that $thread(e) = t$, and $\tau_1|_t = \tau_2|_t$. Since $(\tau_1 e)|_t$ is an extended prefix of $\tau|_t$, then either $(\tau_1 e)|_t$ is a prefix of τ , or $op(e) = read$, and there exists e' , such that $thread(e') = n$, $op(e') = read$, $target(e') = target(e)$, and $\tau_1|_t e'$ is a prefix of $\tau|_t$.

Let e'' be e , if $op(e) \neq read$, or $e'' = e[state(e^w)/state]$, if $e^w = latest_{write}(\tau_2 \upharpoonright_{target(e)})$. Then $\tau_2 e''$ is an extended interleaving prefix. If $op(e) \neq read$, and $\tau_2 e$ is consistent, then it also is a consistent extended interleaving prefix. If $op(e) = read$, then, since $\tau_2 e''$ is consistent (by the choice of e''), the property follows. \square

An important corollary of Theorem 30 is that the feasibility closure does not depend on the legal trace chosen to represent a sequentially consistent trace, as it contains all the representative legal traces.

Corollary 9 *If τ is a consistent interleaving of consistent trace τ' then $feasible(\tau) = feasible(\tau')$.*

19.7 Model Checking the Feasibility Closure

As a corollary to our feasibility closure characterization, we show that our model can be explored computationally by providing a model checking algorithm.

Similarly to happens-before model checking algorithms, the complexity of our algorithm is dominated by the number of causally equivalent feasible traces being explored; however, its coverage is considerably greater, as shown in the companion technical report [79].

We here don't fix any particular type of properties to be checked; we simply assume a generic procedure φ to check whether a given trace satisfies the desired property.

When all events in an execution are recorded, this Since, as discussed in Section 19.4, a feasible trace completely determines the state obtainable upon producing it, the algorithm fully becomes an explicit state model checker, therefore φ could also be used to check state assertions in addition to trace properties.

Model checking the feasibility closure. Algorithm 1 can be used to explore (and check properties against) the feasibility closure of a given trace. It takes as input a trace τ_0 and a procedure φ saying whether a property is satisfied by a (partial) trace (and state), and checks whether all traces in the feasibility closure of τ_0 (and their corresponding states) satisfy the property of φ .

In the initialization phase, the original trace is split into threads and each thread projection is loaded into a stack, with first events in the thread at top of the stack, and the store initialized with *neverViolate* for both variables and mutexes. We additionally maintain a set of enabled threads (**Advanceable**), that is, threads for which all events generated had the same state as in the original execution, therefore they can still be advanced. The trace created, τ , is also maintained as a stack, but with first events at bottom of the stack; it is initially empty. Variable t keeps track of the index of the thread which should be advanced next. The main loop is a backtracking loop, exiting only when the entire space has been explored. Inside the loop, the first part (lines 3–6) checks whether the next thread can be advanced. If a thread is found, the state is modified accordingly (lines 12–15), disabling further advances to the thread if the state of the added event differs from the one in the observed trace (lines 8–10); note that in the latter case, the top event in the corresponding thread needs not be removed, since the thread is disabled.

Input: Trace τ_0 of size n over k threads.
Maps: $\text{thread} : \{1, \dots, k\} \rightarrow \text{Stack}$
 $\sigma : \text{Locations} \rightarrow \text{Int}$
Initial: $\sigma[x] \leftarrow \text{neverViolate}$, for all variables and mutexes
 $\text{thread}[t] \leftarrow \tau_0|_t$, for all threads
 $\text{Advanceable} \leftarrow \{1, \dots, k\}$

```

1  $\tau \leftarrow \epsilon; t \leftarrow 0;$ 
2 while  $t < k$  do
3    $t \leftarrow t + 1;$ 
4   if  $t \in \text{Advanceable}$  then
5      $e \leftarrow \text{top}(\text{thread}[t]);$ 
6     if  $(\text{op}(e) \neq \text{acquire} \vee \sigma[\text{target}(e)] = \text{neverViolate}) \wedge$ 
        $(\text{op}(e) \neq \text{read} \vee \sigma[\text{target}(e)] \neq \text{neverViolate})$  then // advance
7        $l \leftarrow \text{target}(e);$ 
8       if  $\text{op}(e) = \text{read} \wedge \text{state}(e) \neq \sigma[l]$  then // extended prefix
9          $\text{Advanceable} \leftarrow \text{Advanceable} \setminus \{t\};$ 
10         $\text{state}(e) \leftarrow \sigma[l];$ 
11      else // update state
12         $\text{pop}(\text{thread}[t]);$ 
13        if  $\text{op}(e) = \text{write}$  then  $\sigma[l] \leftarrow \text{state}(e);$ 
14        ;
15        if  $\text{op}(e) = \text{acquire}$  then  $\sigma[l] \leftarrow \sigma[l] - 1;$ 
16        ;
17        if  $\text{op}(e) = \text{release}$  then  $\sigma[l] \leftarrow \sigma[l] + 1;$ 
18        ;
19      end
20       $\text{push}(\tau, e);$  check  $\tau$  against  $\varphi;$ 
21       $t \leftarrow 0;$ 
22    end
23  end
24  while  $t = k \wedge \tau \neq \epsilon$  do // backtrack
25     $e \leftarrow \text{pop}(\tau); t \leftarrow \text{thread}(e); l \leftarrow \text{target}(e);$ 
26    if  $t \notin \text{Advanceable}$  then // extended prefix
27       $\text{Advanceable} \leftarrow \text{Advanceable} \cup \{t\};$ 
28    else // restore state
29       $\text{push}(\text{thread}[t], e);$ 
30      if  $\text{op}(e) = \text{write}$  then  $\sigma[l] \leftarrow \text{state}(\text{latest}(\tau|_l));$ 
31      ;
32      if  $\text{op}(e) = \text{acquire}$  then  $\sigma[l] \leftarrow t;$ 
33      ;
34      if  $\text{op}(e) = \text{release}$  then  $\sigma[l] \leftarrow \text{neverViolate};$ 
35      ;
36    end
37  end
38 end

```

Algorithm 1: Model checking the feasibility closure

Then, τ is advanced and added to the result set, property φ is checked (line 17), and the search for the next advance-able thread is restarted (line 18). If no additional thread can be advanced from this state, the algorithm backtracks, undoing the effects of previous advances (lines 21–31).

A simple amortized analysis of Algorithm 1 shows that, without any additional knowledge about the property to check φ , it essentially performs a minimal amount of work: it generates and checks against φ each consistent extended interleaving prefix of τ_0 , searching for each next event through the tops of at most k thread stacks. Supposing that φ is a simple safety property taking constant time and memory to evaluate in any given state σ , which is frequently the case in many situations, the time complexity of our algorithm is $\mathcal{O}(|feasible(\tau_0)| \times k)$ and its memory complexity is $\mathcal{O}(|\tau_0|)$; recall that $feasible(\tau_0)$ is prefix-closed.

Happens-before based model checkers can exploit the property being checked or the structure of the program to gain efficiency (but not coverage). We envision similar techniques could potentially be applied to our models. However, here we are not trying to propose an *optimal* model checker but rather to show that the maximum model is algorithmically analyzable, not just an existential entity, and it can be the basis on which other analysis techniques can be built.

19.8 Related Work and Discussion

Beginning with the introduction of the Happens-Before ordering by Lamport [179], there has been a considerable amount of research on models and techniques to abstract executions for the purpose of inferring causally equivalent executions satisfying/violating particular but important properties, such as dataraces or atomicity/serializability [236, 27, 207, 243, 232, 102, 141, 266, 263, 267]. Section 19.5 shows that the sound causal models upon which the above mentioned techniques were based [194, 267, 243, 236, 207] are subsumed by the maximal causal model; their soundness follows as a corollaries of Theorem 28.

Ganai and Gupta [104] apply a similar technique for software model checking, attempting to reduce the state space to be explored using sequential consistency constraints. Similarly, building on a previous draft of this paper, Said et al. [231] encode the axioms of our proposed model (extended with constructs for thread creation and wait/notify) into an SMT solver and use that to effectively search the model for potential dataraces in Java programs.

Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [206], or to use information about the program and about the property to be checked to further relax the models of executions [63, 266].

Adding or removing attributes from events. Our choice of which attributes to be included in an event was based on the idea of observing the execution of any multithreaded program executed on any machine offering no guarantees other than sequential consistency. Therefore, no semantical information is assumed about the program other than the identity of the thread performing an operation. There are other possible choices, each with their benefits. For example, Sinha et al. [246] choose not to record the value read/written in the memory. Thus, their model must preserve the read-after-write dependence, and the set of comprised traces is thus comparable with that of Wand and Stoller [267], and Sen et al. [243]. We believe a similar maximality result could be proved for traces of this kind, but that has not been attempted yet. In contrast, Wang et al. [266] enrich the events with symbolic information extracted from the program executing them. This allows them to obtain more comprehensive models at the expense of having to analyze the code. Since analyzing the code statically leads quickly to undecidability issues, and thus static analyzers need to be conservative, we believe there might indeed be no similar maximality result for these types of models, their coverage increasing with the power of the analysis.

Causal properties of traces. Since our model associates for a trace all traces which can be obtained by all programs which can obtain that trace, this allows for program-independent definitions of causal properties. For example, Wang and Stoller [267] propose serializability of a trace τ as the property that there exists an alternative execution of the program producing an interleaving of τ in which each transaction is a sequential block. Farzan and Madhusudan [98] relax this constraint by requiring that for each transaction there exists an alternative execution of the program producing an interleaving of τ containing that transaction as a sequential block. Sen et al. [243] say that a trace exhibits a datarace if there exists an alternative execution of the program producing an (partial) interleaving of τ in which the conflicting events are consecutive.

The program-independent properties associated to any of the above (program-dependent) definitions can be obtained by simply replacing the

(rather informal) “alternative execution of the program producing an interleaving of τ ” with “a τ -feasible trace”, as defined by Definition 113. Formal definitions of these causal properties can be found in the companion technical report [79].

19.9 Conclusion

We have shown that, by axiomatizing basic properties of (sequentially consistent) concurrent systems, one can obtain *maximally sound causal models* for concurrent executions, which can be naturally associated to each observed trace, capturing all feasible traces which could be inferred from it. The maximality result has two important theoretical implications. First, verifying the soundness claims for any causal model is reduced to proving that it is a submodel of the maximal one. Second, since the maximal model captures all causally equivalent traces, it allows for universal, program-independent definitions for causal properties. Although this paper focuses on proving the maximality claim of our model, the companion technical report [79] additionally provides a constructive characterization of the proposed model, as well as a model checking algorithm.

Chapter 20

Static Analysis to Improve Runtime Verification

Chapter 21

Semantics-Based Runtime Verification

21.1 Defining a Formal Semantics

21.2 Semantics-Based Symbolic Execution

21.3 Program Verification as Exhaustive Runtime Verification

Chapter 22

Conclusion and Future Work

22.1 Safety Properties and Monitoring

Chapters 3 and 4 presented a comprehensive study of safety properties and of their monitoring, using a uniform formalism and notation. Technically, there were two novel contributions. First, it introduced the notion of a *persistent* safety property, which is the finite-trace correspondent of an infinite-trace safety property, and used it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Second, it rigorously defined the problem of monitoring a safety property, and it showed that it can be arbitrarily hard. These results established a firm foundation for studying safety properties and corresponding monitors and algorithms for various domains of interest, where requirements can be expressed using domain-specific formalisms, such as future-time and past-time temporal logics, context-free grammars, push-down automata, and so on.

Bibliography

- [1] JavaMOP. <http://javamop.com>.
- [2] MOP website, LTL plugin. <http://fsl.cs.uiuc.edu/mop/logic-plugins/ltl>.
- [3] SPECjvm 2008. <http://www.spec.org/jvm2008/>.
- [4] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.
- [5] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
- [6] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
- [7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986. pages 215-246.
- [8] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 345–364. ACM Press, 2005.
- [9] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*, 2005.
- [10] J. Allen. Towards a general theory of actions and time. *Artificial Intelligence*, 23(2):123–154, 1984.
- [11] Bowen Alpern and Fred B. Schneider. Defining liveness. *IPL*, 21(4):181–185, 1985.

- [12] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.
- [13] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
- [14] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.
- [15] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *POPL*, pages 189–198, 1987.
- [16] Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’08)*, pages 143–162. ACM, 2008.
- [17] C. Artho, A. Biere, and K. Havelund. High-level data races. In *The 1st International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS’03)*, April 2003.
- [18] Cyrille Artho, Doron Drusinsky, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. Experiments with Test Case Generation and Runtime Analysis. In *Proc. of ASM’03: Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 87–107, Taormina, Italy, March 2003. Springer.
- [19] Cyrille Artho, Doron Drusinsky, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Păsăreanu, Grigore Roşu, Willem Visser, and Rich Washington. Automated Testing using Symbolic Execution and Temporal Monitoring. *Theoretical Computer Sci.*, to appear, 2005.
- [20] Formal System Laboratory at UIUC. ptCaRet MOP Logic Plugin. <http://fsl.cs.uiuc.edu/index.php/Special:JavaMOPPTCARETOnline>.
- [21] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *TOCS*, 12:91–122, May 1994.
- [22] P. Avgustinov and C. Church. *Trace Monitoring with Free Variables*. PhD thesis, Oxford University, 2009.
- [23] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. ABC: an extensible AspectJ compiler. In *AOSD’05*, pages 87–98. ACM, 2005.

- [24] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitoring feasible. In Richard P. Gabriel, editor, *OOPSLA'07*. ACM, 2007.
- [25] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitors Feasible. In *OOPSLA'07*, 2007.
- [26] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Genova, Italy, April 2001.
- [27] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *PADTAD'06*, pages 69–78, New York, 2006. ACM.
- [28] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of VMCAI'04*, volume 2937 of *LNCS*, pages 44–57, 2004.
- [29] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.
- [30] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57, 2004.
- [31] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Logic Computation*, pages exn076+, November 2008.
- [32] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, 2007.
- [33] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. Monopoly: Monitoring usage-control policies. In *Runtime Verification (RV'11)*, pages 360–364, 2011.
- [34] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20, 2011.
- [35] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Runtime Verification (RV'10)*, pages 168–182, 2010.

- [36] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190. ACM, 2006.
- [37] E. Bodden, Patrick Lam, and Laurie Hendren. Tracematches Benchmarks, 2008. <http://abc.comlab.ox.ac.uk/tmahead>.
- [38] Eric Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
- [39] Eric Bodden. J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
- [40] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.
- [41] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549, 2007.
- [42] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, pages 183–197. Springer, 2010.
- [43] Eric Bodden and Volker Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In *Software Composition*, pages 147–162, 2006.
- [44] Ronald V. Book and Friedrich Otto. *String-rewriting systems*. Texts and monographs in computer science. Springer, 1993.
- [45] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236:35–132, 2000.
- [46] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [47] J R Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. Logic, Methodology and Philosophy of Sciences. Stanford University Press, 1962.

- [48] D. Calvanese, G. De Giacomo, and M. Y. Vardi. Reasoning about actions and planning in LTL action theories. In *KR*, pages 593–602, 2002.
- [49] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
- [50] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
- [51] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In *14th Workshop on Model Checking Software (SPIN)*, volume 4595 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2007. Tool paper.
- [52] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In *Model Checking Software (SPIN’07)*, volume 4595 of *LNCS*, pages 279–283, 2007.
- [53] F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proc. of RV’03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 106–125, Boulder, Colorado, USA, 2003. Elsevier Science.
- [54] Feng Chen, Marcelo d’Amorim, and Grigore Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of ICFEM’04*, volume 3308 of *LNCS*, pages 357–372, 2004.
- [55] Feng Chen, Marcelo D’Amorim, and Grigore Roşu. A formal monitoring-based framework for software development and analysis. In *ICFEM’04*, volume 3308 of *LNCS*, pages 357–372, 2004.
- [56] Feng Chen, Marcelo D’Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *RV’05*, volume 144(4) of *ENTCS*, 2005.
- [57] Feng Chen, Patrick Meredith, Dongyun Jin, and Grigore Rosu. Efficient formalism-independent monitoring of parametric properties. In *IEEE/ACM International Conference on Automated Software Engineering (ASE’09)*, pages 383–394, 2009.
- [58] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specif. and Implementation. In *RV’03*, volume 89(2) of *ENTCS*, 2003.
- [59] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS’05*, volume 3440 of *LNCS*, pages 546–550, 2005.

- [60] Feng Chen and Grigore Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
- [61] Feng Chen and Grigore Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2006-2965, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.
- [62] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA'07*, 2007.
- [63] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *CAV'07*, volume 4590 of *LNCS*, pages 240–253, 2007.
- [64] Feng Chen and Grigore Roşu. Mining Parametric State-Based Specifications from Executions. Technical Report UIUCDCS-R-2008-3000, Dept. of Computer Science at UIUC, 2008.
- [65] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
- [66] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. Effective predictive runtime analysis using sliced causality and atomicity. Technical Report UIUCDCS-R-2007-2905, University of Illinois at Urbana-Champaign, Department of Computer Science, 2007.
- [67] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 221–230, New York, NY, USA, 2008. ACM.
- [68] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [69] M. Clavel. The ITP Tool. In *Logic, Language and Information. Proc. of the First Workshop on Logic and Language*, pages 55–62. Kronos, 2001.
- [70] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In *3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
- [71] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

- [72] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
- [73] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual. <http://maude.cs.uiuc.edu>.
- [74] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude System. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.
- [75] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. A Maude Tutorial, March 2000. Manuscript at <http://maude.csl.sri.com/papers>.
- [76] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002.
- [77] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM’09)*, pages 33–37, 2009.
- [78] James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of ICSE’00: International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
- [79] Traian Florin Șerbănuță, Feng Chen, and Grigore Roșu. Maximal causal models for sequentially consistent systems. Technical Report <http://hdl.handle.net/2142/27708>, University of Illinois at Urbana-Champaign, October 2011.
- [80] M. Dahm. BCEL. <http://jakarta.apache.org/bcel>.
- [81] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [82] M. D’Amorim and G. Roșu. Efficient monitoring of omega-languages. In *CAV’05*, volume 3576 of *LNCS*, pages 364–378. Springer, July 2005.

- [83] Marcelo d’Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
- [84] Marcelo d’Amorim and Grigore Roşu. Efficient monitoring of ω -languages. In *Proceedings of 17th International Conference on Computer-aided Verification (CAV’05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364 – 378. Springer, 2005.
- [85] Marcelo d’Amorim and Grigore Roşu. Efficient monitoring of ω -languages. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 364–378, 2005.
- [86] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
- [87] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
- [88] Doron Drusinsky. Temporal Rover. <http://www.time-rover.com>.
- [89] Doron Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV’03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
- [90] A. G. Duncan. Test grammars: A method for generating program test data. In *Workshop on Software Testing and Test Documentation*, pages 270–281, 1978.
- [91] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *International Conference on Software Engineering (ICSE’81)*, pages 170–178, 1981.
- [92] Matthew Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis. In *Runtime Verification (RV’10)*, volume 6418 of *LNCS*, pages 36–50. Springer, 2010.
- [93] Matthew Dwyer, Rahul Purandare, and Suzette Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis. In *Runtime Verification (RV’10)*, volume 6418 of *LNCS*, pages 36–50. Springer, 2010.

- [94] Javamop generic parametric monitoring results. <http://fsl.cs.uiuc.edu/index.php/JavaMOP> Experiment Enable Sets.
- [95] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *Symposium on Security and Privacy (SP'00)*, pages 246–. IEEE, 2000.
- [96] Kousha Etessami and Gerard Holzmann. Optimizing Büchi Automata. In *Proc. of Int. Conf. on Concurrency Theory*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000.
- [97] James Ezick. An Optimizing Compiler for Batches of Temporal Logic Formulas. In *Proceedings of ISSTA'04*, pages 183–194. ACM Press, 2004.
- [98] Azadeh Farzan and Parthasarathy Madhusudan. Causal atomicity. In *CAV'06*, volume 4144 of *LNCS*, pages 315–328, 2006.
- [99] C. J. Fidge. Partial Orders for Parallel Debugging. In *Proc. of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, pages 183–194. ACM, 1988.
- [100] B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proc. of RV'01: The First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.
- [101] Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. Collecting Statistics over Runtime Executions. In *Proc. of RV'02: The Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2002. Elsevier.
- [102] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*, pages 256–267, 2004.
- [103] Dov M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer, 1989.
- [104] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN'08*, volume 5156 of *LNCS*, pages 114–133, 2008.
- [105] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(1):1203–1233, September 2000.

- [106] Michael Garey. Optimal Binary Identification Procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, 1972.
- [107] P. Gastin and D. Oddoux. LTL with Past and Two-Way Very-Weak Alternating Automata. In *Proceedings of MFCS'03*, number 2747 in LNCS, pages 439–448. Springer, 2003.
- [108] Marc Geilen. On the Construction of Monitors for Temporal Logic Properties. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.
- [109] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proc. of the 15th Workshop on Protocol Specification, Testing, and Verification*. North-Holland, 1995.
- [110] M. Ghallab and A.M. Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *IJCAI*, pages 1297–1303, 1989.
- [111] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [112] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. of POPL'97: the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.
- [113] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, Automated Software Engineering '00*, pages 123–131. IEEE, 2000. (Grenoble, France).
- [114] J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science, to appear, Frauenchiemsee, Germany, September 2002. Springer-Verlag.
- [115] Joseph Goguen, Kai Lin, Grigore Roşu, Akira Mori, and Bogdan Warinschi. An overview of the Tatami project. In *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 61–78. Elsevier, 2000.
- [116] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.
- [117] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

- [118] Simon Goldsmith, Robert O’Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA’05*, pages 385–402. ACM, 2005.
- [119] Elsa Gunter and Doron Peled. Tracing the Executions of Concurrent Programs. In *Proc. of RV’02: Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Copenhagen, Denmark, 2002. Elsevier.
- [120] Elsa L. Gunter, Robert P. Kurshan, and Doron Peled. PET: An Interactive Software Testing Tool. In *Proc. of CAV’00: Computer Aided Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 552–556, Chicago, Illinois, USA, 2003. Springer-Verlag.
- [121] Elsa L. Gunter and Doron Peled. Using Functional Languages in Formal Methods: The PET System. In *Parallel and Distributed Processing Techniques and Applications*, pages 2981–2986. CSREA, 2000.
- [122] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Programming languages and analysis for security (PLAS’08)*, pages 11–20. ACM, 2008.
- [123] Kevin W. Hamlen, J. Gregory Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.
- [124] K.V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.
- [125] K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
- [126] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV’01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [127] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE’01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [128] K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV’02)* satellite workshop.

- [129] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Technology Transfer*, 6(2):158–173, 2004. (also TACAS’02, LNCS 2280).
- [130] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS’02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
- [131] Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and Error Pattern Based Program Monitoring. In *Proc. of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands, October 2001.
- [132] Klaus Havelund, Scott Johnson, and Grigore Roşu. Specification and Error Pattern Based Program Monitoring. In *European Space Agency Workshop on On-Board Autonomy*, Noordwijk, The Netherlands, 2001.
- [133] Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001. An earlier version occurred in the Proc. of SPIN’98: the fourth SPIN workshop, Paris, France, 1998.
- [134] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue containing selected submissions to SPIN’98: the fourth SPIN workshop, Paris, France, 1998.
- [135] Klaus Havelund and Grigore Roşu. Java PathExplorer – A Runtime Verification Tool. In *Proc. of i-SAIRAS’01: the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Montreal, Canada, June 2001.
- [136] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. of RV’01: the First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
- [137] Klaus Havelund and Grigore Roşu. *Workshops on Runtime Verification (RV’01, RV’02, RV’04)*, volume 55, 70(4), to appear of *ENTCS*. Elsevier, 2001, 2002, 2004.
- [138] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’02)*, volume 2280 of *LNCS*, pages 342–356. Springer, April 2002.

- [139] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *STTT*, 6(2):158–173, 2004.
- [140] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Proc. of FME’96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, England, 1996. Springer.
- [141] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *TPDS*, 4(7):827–840, 1993.
- [142] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.
- [143] S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
- [144] Gerard Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [145] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
- [146] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. of ICSE’99: International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
- [147] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- [148] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [149] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. Addison-Wesley, 2001.
- [150] B. Houssais. Verification of an algol 68 implementtion. In *Strathclyde Algol 68 Conference*, 1977.
- [151] Jieh Hsiang. *Refutational Theorem Proving using Term Rewriting Systems*. PhD thesis, University of Illinois at Champaign-Urbana, 1981.
- [152] Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.

- [153] Jeff Huang, Qingzhou Luo, and Grigore Rosu. Gpredict: Generic predictive concurrency analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. ACM, 2015.
- [154] Jeff Huang, Patrick Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*, pages 337–348. ACM, June 2014.
- [155] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. In *International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 39–49. ACM, 2007.
- [156] Laurent Hyafil and Ronald L. Rivest. Computing Optimal Binary Decision Trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
- [157] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In *Proceedings of STACS'03*, volume 2607 of *LNCS*, pages 179–190, 2003.
- [158] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In H. Alt and M. Habib, editors, *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer Science (STACS '03)*, volume 2607 of *Lecture Notes in Computer Science*, page 179. Springer-Verlag, Berlin, 2003.
- [159] JavaCC. Url. http://www.webgain.com/products/java_cc.
- [160] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, June 2011.
- [161] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. Technical Report <http://hdl.handle.net/2142/18751>, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011.
- [162] JTrek. Web page. <http://www.compaq.com/java/download>.
- [163] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A Decision Algorithm for Full Propositional Temporal Logic. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 97–109. Springer, 1993.
- [164] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353, 2001.

- [165] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- [166] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
- [167] J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.
- [168] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
- [169] David Kortenkamp, Tod Milam, Reid Simmons, and Joaquin Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proc. of RV'01: First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.
- [170] A.A. Krokhin, P. Jeavons, and P. Jonsson. Reasoning about temporal relations: The tractable subalgebras of Allen's interval algebra. *J. ACM*, 50(5):591–640, 2003.
- [171] O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From Linear-Time to Branching-Time. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.
- [172] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proc. of CAV'99: Conference on Computer-Aided Verification*, Trento, Italy, 1999.
- [173] O. Kupferman and S. Zuhovitzky. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In *Proc. of the International Symposium on Mathematical Foundations of Computer Science*, volume 2420 of *Lecture Notes in Computer Science*, 2002.
- [174] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *Proc. of MFCS'02*, volume 2420 of *LNCS*, pages 446–458, 2002.
- [175] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
- [176] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, October 2001.

- [177] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19:291–314, October 2001.
- [178] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila. Autonomous rover navigation on unknown terrains, functions and integration. *International Journal of Robotics Research*, 2003.
- [179] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
- [180] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
- [181] Leslie Lamport. Logical foundation. In M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, F. B. Schneider, M. Paul, and H. J. Siegert, editors, *Distributed systems: Methods and tools for specification. An advanced course*, volume 190 of *LNCs*, pages 119–130. Springer-Verlag, 1985.
- [182] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, 2000.
- [183] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE’11)*, pages 591–600. ACM, 2011.
- [184] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [185] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
- [186] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [187] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
- [188] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
- [189] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling lscs into aspectj. In *FSE’06*, pages 219–230, 2006.
- [190] Nicolas Markey. Temporal Logic with Past is Exponentially more Succinct. *EATCS Bulletin*, 79:122–128, 2003.

- [191] Nicolas Markey and Philippe Schnoebelen. Model Checking a Path (Preliminary Report). In *Proc. of CONCUR'03: International Conference on Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 251–265, Marseille, France, August 2003. Springer.
- [192] Michael Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'07*, pages 365–383. ACM, 2005.
- [193] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.
- [194] A Mazurkiewicz. Trace theory. In *Advances in Petri nets*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag.
- [195] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.
- [196] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.
- [197] Patrick Meredith and Grigore Roşu. Runtime verification with the RV system. In *First International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lecture Notes in Computer Science*, pages 136–152. Springer, 2010.
- [198] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the mop runtime verification framework. *Journal on Software Tools for Technology Transfer (J. of STTT)*, 2010. To appear.
- [199] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. <http://dx.doi.org/10.1007/s10009-011-0198-6>.
- [200] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
- [201] José Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proc. of WADT'97: Workshop on Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61, Tarquinia, Italy, June 1998. Springer.
- [202] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, 34:129–153, 1956.

- [203] Bernard Moret. Decision Trees and Diagrams. *ACM Comp. Surv.*, 14(4):593–623, 1982.
- [204] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC'01)*, June 2001.
- [205] G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
- [206] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI'06*, pages 308–319, 2006.
- [207] Robert O’Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.
- [208] Dennis Oddoux and Paul Gastin. LTL2BA. <http://www.liafa.jussieu.fr/~oddoux/ltl2ba/>.
- [209] T. O’Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.
- [210] Eclipse Org. Aspectj project. <http://eclipse.org/aspectj/>.
- [211] David Y.W. Park, Ulrich Stern, and David L. Dill. Java Model Checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.
- [212] Amir Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
- [213] P.A. Purdom. A sentence generator for testing parsers. *BIT*, 2:336–375, 1972.
- [214] D. J. Richardson, S. L. Aha, and T. O. O’Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.
- [215] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
- [216] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic – translation to LTL and monitor synthesis. Technical Report UIUCDCS-R-2006-2681, University of Illinois at Urbana-Champaign, January 2006.
- [217] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *J. of Automated Software Engineering*, 12(2):151–197, 2005.

- [218] G. Roşu and K. Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. Technical Report TR 01-08, NASA - RIACS, May 2001.
- [219] G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *RTA '03*, volume 2706 of *LNCS*. Springer, 2003.
- [220] Grigore Roşu. On Safety Properties and Their Monitoring. Technical Report UIUCDCS-R-2007-2850, Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, 2007.
- [221] Grigore Roşu and Feng Chen. Parametric Trace Slicing and Monitoring. Technical Report UIUCDCS-R-2008-2977, University of Illinois at Urbana-Champaign, 2008.
- [222] Grigore Roşu and Klaus Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. RIACS Technical report TR 01-08, January 2001.
- [223] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
- [224] Grigore Roşu and Klaus Havelund. Rewriting-Based Techniques for Runtime Verification. *Journal of Automated Software Engineering*, 2005. to appear.
- [225] Grigore Roşu. K: a rewrite-based framework for modular lang. design, semantics, analysis and implementation (V2). Technical Report UIUCDCS-R-2006-2802, 2006.
- [226] Grigore Rosu. An effective algorithm for the membership problem for extended regular expressions. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'07)*, volume 4423 of *LNCS*, pages 332–345. Springer-Verlag, 2007.
- [227] Grigore Rosu. On safety properties and their monitoring. *Sci. Ann. Comp. Sci.*, 22(2):327–365, 2012.
- [228] Grigore Roşu and Feng Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1):1–47, Feb 2012. Short version presented at TACAS 2009.
- [229] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.

- [230] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer-Verlag, 1998.
- [231] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods'11*, volume 6617 of *LNCS*, pages 313–327, 2011.
- [232] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *TOCS*, 15(4):391–411, 1997.
- [233] Stefan Savage, Michael Burrows, Greg Nelson, Patrik Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
- [234] Fred B. Schneider. *On Concurrent Programming*. Springer, 1997.
- [235] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
- [236] Edith Schonberg. On-the-fly detection of access anomalies. *Best of PLDI 1979-1999*, 39:313–327, April 2004.
- [237] Alper Sen and Vijay K. Garg. Partial Order Trace Analyzer (POTA) for Distrubted Programs. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, Boulder, Colorado, USA, 2003. Elsevier Science.
- [238] K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proc. of RV'03: the Third International Workshop on Runtime Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 162–181, Boulder, Colorado, USA, 2003. Elsevier Science.
- [239] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urbana Champaign, April 2003.
- [240] Koushik Sen, Grigore Roşu, and Gul Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proc. of ESEC/FSE'03: European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, Helsinki, Finland, September 2003.

- [241] Koushik Sen and Grigore Rosu. Generating optimal monitors for extended regular expressions. In *Proceedings of 3rd International Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, pages 226–245. Elsevier, June 2003.
- [242] Koushik Sen, Grigore Roşu, and Gul Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 123–138. Springer, 2002.
- [243] Koushik Sen, Grigore Roşu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS'05*, volume 3535 of *LNCS*, pages 211–226, 2005.
- [244] Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In Shaz Qadeer, editor, *Runtime Verification (RV'12)*, Lecture Notes in Computer Science, 2012. To appear.
- [245] Natarajan Shankar, Sam Owre, and John M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
- [246] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *MEMOCODE'11*.
- [247] E. Sirer and B.N. Bershad. Using production grammars in software testing. In *Domain Specific Languages (DSL'00)*, pages 1–13, 1999.
- [248] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.
- [249] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.
- [250] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.
- [251] L. A. Smith, J. M. Bull, and J. Obdržálek. A parallel java grande benchmark suite. In *Supercomputing (SC'01)*, pages 8–8. ACM, 2001.
- [252] O. Sokolsky and M. Viswanathan. *Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
- [253] Soot website. <http://www.sable.mcgill.ca/soot/>.
- [254] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *STOC*, pages 1–9. ACM Press, 1973.

- [255] Larry Joseph Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Massachusetts Institute of Technology, 1974.
- [256] Scott D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244, Stanford, California, USA, 2000. Springer.
- [257] Volker Stolz. Temporal assertions with parameterized propositions. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin / Heidelberg, 2007.
- [258] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Runtime Verification (RV'05)*, volume 144 of *ENTCS*, pages 109–124. Elsevier, 2005.
- [259] Robert E. Strom and Shaula Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.
- [260] Robert E. Strom and Shaula Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, January 1986.
- [261] Deian Tabakov and Moshe Vardi. Optimized temporal monitors for systemc. In *First International Conference on Runtime Verification (RV'10)*, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010.
- [262] K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
- [263] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pages 334–345, 2006.
- [264] M. Vilain, H. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, Los Altos, CA, 1989.
- [265] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proc. of ASE'00: International Conference on Automated Software Engineering*, Grenoble, France, September 2000. IEEE CS Press.
- [266] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM'09*, pages 256–272, 2009.

- [267] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP'06*, pages 137–146, 2006.
- [268] Pierre Wolper. Constructing Automata from Temporal Logic Formulas: a Tutorial. volume 2090 of *LNCS*, pages 261–277. Springer, 2002.
- [269] H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proc. of MFCS'00*, volume 1893 of *LNCS*, pages 699–708, 2000.
- [270] H. Yamamoto. A new recognition algorithm for extended regular expressions. In *Proceedings of ISAAC'01*, volume 2223 of *LNCS*, pages 257–267, 2001.
- [271] H. Yamamoto and T. Miyazaki. A fast bit-parallel algorithm for matching extended regular expressions. In *Proc. of COCOON'03*, volume 2697 of *LNCS*, pages 222–231, 2003.